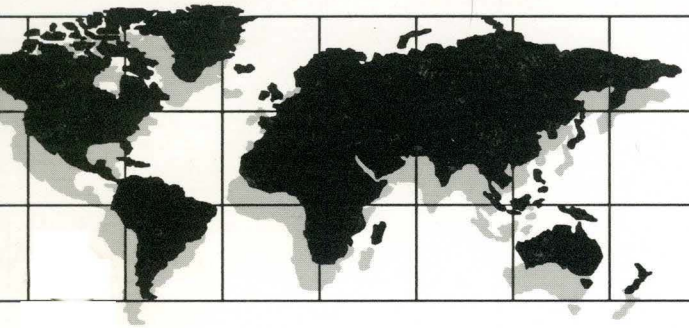

CTOS

Procedural Interface

Reference Manual

Volume 1
Operations A through G

UNISYS

UNISYS

CTOS®
Procedural Interface
Reference Manual
Volume 1
Operations A through G

Copyright © 1990, 1991, 1992 Unisys Corporation
All Rights Reserved
Unisys is a trademark of Unisys Corporation

This volume has been updated for CTOS III with pages from Update 1, 4357 4714-110

CTOS III 1.0
CTOS II 3.4
CTOS I 3.4
CTOS/XE 3.4
Development Utilities 12.2
Standard Software12.2
Video Access Method 4.0
Priced Item

August 1992

Printed in USA
4357 4714-100

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information or software material, including direct, indirect, special or consequential damages.

You should be careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED RIGHTS LEGEND. Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.227-7013 and FAR 52.227-14 for commercial computer software.

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

Convergent, Convergent Technologies, CTOS, NGEN, and SuperGen are registered trademarks of Convergent Technologies, Inc.

ClusterCard, ClusterShare, Context Manager, Context Manager/VM, CT-DBMS, CT-MAIL, CT-Net, CTOS/VM, CTOS/Vpc, Document Designer, Generic Print System, PC Emulator, Print Manager, Series 186, Series 286, Series 386, Series 286i, Series 386i, shared resource processor, SRP, TeleCluster, Voice/Data Services, Voice Processor, X-Bus, and X-Bus+ are trademarks of Convergent Technologies, Inc.

OFIS is a registered trademark of Unisys Corporation.

BTOS is a trademark of Unisys Corporation.

AT, IBM, IBM PC, and OS/2 are registered trademarks of International Business Machines Corporation. IBM PC-AT, IBM PC-XT, and IBM PS/2 are trademarks of International Business Machines Corporation. Intel is a registered trademark of Intel Corporation. MS-DOS and Microsoft are registered trademarks of Microsoft Corporation. UNIX is a registered trademark of AT&T.

Page Status

Volume 1

Page	Issue
Cover	August 1992
i through xxviiB	August 1992
xxviii	Blank
xxix through xliC	August 1992
1-1 through 1-2	Original
2-1 through 2-18B	August 1992
3-1 through 3-6	Original
3-7 through 3-8	August 1992
3-9 through 3-20	Original
3-20A through 3-20D	August 1992
3-21 through 3-22	Original
3-22A through 3-22B	August 1992
3-23 through 3-58	Original
3-59 through 3-62	August 1992
3-63 through 3-70	Original
3-71 through 3-78	August 1992
3-79 through 3-80	Original
3-81 through 3-82	August 1992
3-83 through 3-278	Original
3-279 through 3-280	August 1992
3-281 through 3-314	Original
3-315 through 3-316	August 1992
3-317 through 3-324	Original
3-324A through 3-324B	August 1992
3-325 through 3-326	Original
3-326A through 3-326B	August 1992
3-327 through 3-336	Original
3-336A through 3-336B	August 1992
3-337 through 3-366	Original
3-367 through 3-376	August 1992
3-377 through 3-392	Original

Page	Issue
3-393 through 3-394	August 1992
3-395 through 3-396	Original
3-397 through 3-398	August 1992
3-399 through 3-412	Original
3-413 through 3-418	August 1992
3-419 through 3-426	Original
3-427 through 3-428	August 1992
3-429 through 3-434	Original
3-435 through 3-436	August 1992
3-437 through 3-454	Original
3-454A through 3-454F	August 1992
3-455 through 3-484	Original
3-485 through 3-486B	August 1992
3-487 through 3-510	Original
3-511 through 3-520D	August 1992
3-521 through 3-536	Original
3-536A through 3-536B	August 1992
3-537 through 3-548	Original
3-548A through 3-548B	August 1992
3-549 through 3-550	Original
3-550A through 3-550B	August 1992
3-551 through 3-580	Original
3-580A through 3-580D	August 1992
3-581 through 3-582	August 1992
3-583 through 3-594	Original
3-595 through 3-596	August 1992
3-597 through 3-606	Original
3-606A through 3-606D	August 1992
3-607 through 3-614	Original
3-615 through 3-620	August 1992
3-621 through 3-636	Original
3-637 through 3-638	August 1992
3-639 through 3-682	Original
3-683 through 3-684	August 1992
3-685 through 3-698	Original

Volume 2

Page	Issue
Volume 2 cover	August 1992
i through xxviiB	August 1992
xxviii	Blank
3-699 through 3-710	Original
3-711 through 3-718	August 1992
3-719 through 3-740	Original
3-740A through 3-740P	August 1992
3-741 through 3-742	August 1992
3-743 through 3-744	Original
3-745 through 3-746	August 1992
3-747 through 3-754	Original
3-755 through 3-756B	August 1992
3-757 through 3-762	Original
3-763 through 3-764	August 1992
3-765 through 3-786	Original
3-786A through 3-786B	August 1992
3-787 through 3-792	Original
3-793 through 3-796	August 1992
3-797 through 3-800	Original
3-801 through 3-802	August 1992
3-803 through 3-806	Original
3-807 through 3-808	August 1992
3-809 through 3-810	Original
3-811 through 3-812	August 1992
3-813 through 3-826	Original
3-827 through 3-832B	August 1992
3-833 through 3-838	Original
3-839 through 3-840	August 1992
3-841 through 3-842	Original
3-842A through 3-842H	August 1992
3-843 through 3-846	Original
3-847 through 3-870	August 1992
3-871 through 3-872	Original
3-873 through 3-880	August 1992
3-881 through 3-890	Original
3-891 through 3-894	August 1992
3-895 through 3-924	Original
3-925 through 3-926	August 1992
3-927 through 3-930	Original
3-931 through 3-932	August 1992
3-933 through 3-940	Original

Page	Issue
3-941 through 3-948	August 1992
3-949 through 3-950	Original
3-951 through 3-958	August 1992
3-959 through 3-960	Original
3-961 through 3-966	August 1992
3-967 through 3-1024	Original
3-1025 through 3-1026	August 1992
3-1027 through 3-1036	Original
3-1037 through 3-1038	August 1992
3-1039 through 3-1042	Original
3-1043 through 3-1044	August 1992
3-1045 through 3-1076	Original
3-1077 through 3-1080	August 1992
3-1081 through 3-1082	Original
3-1083 through 3-1084	August 1992
3-1085 through 3-1098	Original
3-1099 through 3-1100	August 1992
3-1101 through 3-1118	Original
3-1119 through 3-1122	August 1992
3-1123 through 3-1126	Original
3-1127 through 3-1130	August 1992
3-1131 through 3-1132	Original
3-1133 through 3-1144	August 1992
3-1145 through 3-1156	Original
3-1156A through 3-1160	August 1992
3-1161 through 3-1176	Original
3-1177 through 3-1182	August 1992
3-1183 through 3-1188	Original
3-1189 through 3-1190	August 1992
3-1191 through 3-1192	Original
3-1193 through 3-1196	August 1992
3-1197 through 3-1214	Original
3-1215 through 3-1216	August 1992
3-1217 through 3-1218	Original
3-1219 through 3-1224	August 1992
3-1225 through 3-1228	Original
3-1229 through 3-1230	August 1992
3-1231 through 3-1236	Original
3-1237 through 3-1242	August 1992
3-1243 through 3-1276	Original
3-1276A through 3-1276B	August 1992
3-1277 through 3-1298	Original

Page	Issue
3-1299 through 3-1302	August 1992
3-1303 through 3-1332	Original
3-1333 through 3-1334	August 1992
3-1335 through 3-1342	Original
3-1343 through 3-1344	August 1992
3-1345 through 3-1354	Original
3-1355 through 3-1360	August 1992
3-1361 through 3-1368	Original
3-1368A through 3-1368B	August 1992
3-1369 through 3-1390	Original
3-1390A through 3-1390Z4	August 1992

Volume 3

Volume 3 cover	August 1992
i through xxviiB	August 1992
xxviii	Blank
3-1391 through 3-1396	Original
3-1397 through 3-1402	August 1992
3-1403 through 3-1404	Original
3-1405 through 3-1406	August 1992
3-1407 through 3-1408	Original
3-1409 through 3-1410	August 1992
3-1411 through 3-1412	Original
3-1413 through 3-1414	August 1992
3-1415 through 3-1440	Original
3-1441 through 3-1442	August 1992
3-1443 through 3-1450	Original
3-1451 through 3-1452	August 1992
3-1453 through 3-1454	Original
3-1455 through 3-1458	August 1992
3-1459 through 3-1460	Original
3-1461 through 3-1462	August 1992
3-1463 through 3-1468	Original
3-1468A through 3-1468Z14	August 1992
3-1469 through 3-1494	Original
3-1495 through 3-1496	August 1992
3-1497 through 3-1504	Original
3-1505 through 3-1508	August 1992
3-1509 through 3-1520	Original
3-1521 through 3-1522	August 1992
3-1523 through 3-1528	Original
3-1529 through 3-1530	August 1992

Page	Issue
3-1531 through 3-1532	Original
3-1533 through 3-1534	August 1992
3-1535 through 3-1542	Original
3-1542A through 3-1542D	August 1992
3-1543 through 3-1570	Original
3-1571 through 3-1574	August 1992
3-1575 through 3-1588	Original
3-1589 through 3-1592	August 1992
3-1593 through 3-1598	Original
3-1599 through 3-1602	August 1992
3-1603 through 3-1606	Original
3-1606A through 3-1606B	August 1992
3-1607 through 3-1630	Original
3-1631 through 3-1632	August 1992
3-1633 through 3-1652	Original
3-1653 through 3-1654	August 1992
3-1655 through 3-1658	Original
3-1659 through 3-1660	August 1992
3-1661 through 3-1662	Original
3-1663 through 3-1664	August 1992
3-1665 through 3-1674	Original
3-1675 through 3-1676	August 1992
3-1677 through 3-1720	Original
3-1721 through 3-1818	Original
3-1819 through 3-1822	August 1992
3-1823 through 3-1866	Original
3-1867 through 3-1868	August 1992
3-1869 through 3-1870	Original
3-1871 through 3-1878	August 1992
3-1879 through 3-1888	Original
3-1889 through 3-1890	August 1992
3-1891 through 3-1894	Original

Volume 4

Page	Issue
Volume 4 cover	August 1992
i through xxviiB	August 1992
xxviii	Blank
4-1 through 4-8	August 1992
4-1 through 4-26	Original
4-27 through 4-46B	August 1992
4-47 through 4-66	Original
4-67 through 4-74	August 1992
4-75 through 4-100	Original
4-101 through 4-108	August 1992
4-109 through 4-126	Original
4-127 through 4-134	August 1992
4-135 through 4-138	Original
4-139 through 4-152	August 1992
4-153 through 4-172	Original
4-173 through 4-182	August 1992
4-183 through 4-222	Original
4-222A through 4-228	August 1992
4-229 through 4-242	Original
4-243 through 4-250	August 1992
4-251 through 4-270	Original
4-271 through 4-274	August 1992
4-275 through 4-288	Original
4-289 through 4-290	August 1992
4-291 through 4-292	Original
4-292A through 4-294	August 1992
4-295 through 4-326	Original
4-327 through 4-338	August 1992
4-339 through 4-350	Original
4-351 through 4-352	August 1992
4-353 through 4-386	Original
4-387 through 4-396	August 1992
4-397 through 4-404	Original
4-405 through 4-412	August 1992
4-413 through 4-462	Original
A-1	Original
A-2	Blank
B-1 through B-14	Original
B-15 through B-16	August 1992
C-1 through C-2	August 1992
C-3 through C-4	Original

Page	Issue
C-5 through C-12	August 1992
D-1 through D-6	Original
D-7 through D-8	August 1992
D-9 through D-10	Original
E-1 through E-4	Original
F-1 through F-3	Original
F-4	Blank
G-1 through G-4	Original
G-5 through G-10	August 1992
G-11 through G-15	Original
G-16	Blank
H-1 through H-4	August 1992
H-5 through H-9	Original
H-10	Blank
I-1 through I-2	Original
J-1 through J-31	Original
J-32	Blank
K-1 through K-2	August 1992

Contents

About This Manual

1 Introduction to CTOS Operations 1-1

2 Operations by Function 2-1

3 Operations

AbortRequests 3-3
AccessVersion 3-5
AcquireByteStreamC 3-7
AddQueue 3-11
AddQueueEntry 3-13
AllocAllMemorySL 3-17
AllocAreaSL 3-19
AllocateSegment 3-20A
AllocCommDmaBuffer 3-20C
AllocExch 3-21
AllocHugeMemory 3-22A
AllocMemoryFramesSL 3-23
AllocMemoryInit 3-25
AllocMemoryLL 3-27
AllocMemorySL 3-29
AllocUserNumbers 3-31
AsGetVolume 3-33
AsSetVolume 3-37
AssignKbd 3-41
AssignVidOwner 3-43
AsyncRequest 3-45
AsyncRequestDirect 3-47

AtFileInit 3-49
AtFileNext 3-51
Beep 3-53
BuildAsyncRequest 3-55
BuildAsyncRequestDirect 3-57
BuildFileSpec 3-59
BuildFullSpecFromPartial 3-65
BuildLdtSlot 3-69
BuildSpecFromDir 3-71
BuildSpecFromFile 3-73
BuildSpecFromNode 3-75
BuildSpecFromPassword 3-77
BuildSpecFromVol 3-81
CallRealCommIsr 3-83
CacheClose 3-85
CacheFlush 3-87
CacheGetEntry 3-91
CacheGetStatistics 3-97
CacheGetStatus 3-101
CacheInit 3-103
CacheReleaseEntry 3-107
CdAbsoluteRead 3-111
CdAudioCtl 3-115
CdControl 3-131
CdClose 3-139
CdDirectoryList 3-141
CdGetDirEntry 3-143
CdGetVolumeInfo 3-145
CdOpen 3-149
CdRead 3-153
CdSearchClose 3-155
CdSearchFirst 3-157
CdSearchNext 3-161
CdServiceControl 3-165
CdVerifyPath 3-169
CdVersion 3-171
CfaFfVersion 3-175
CfaServerVersion 3-177

CfaWaVersion 3-179
Chain 3-181
ChangeCommLineBaudRate 3-185
ChangeFileLength 3-189
ChangeOpenMode 3-191
ChangePriority 3-193
ChangeProcessPriority 3-195
ChangeUserNumberRequests 3-197
Check 3-199
CheckContextStack 3-201
CheckErc 3-203
CheckForOperatorRestartC 3-205
CheckpointBs 3-207
CheckpointBsAsyncC 3-209
CheckPointBsC 3-211
CheckpointRsFile 3-213
CheckpointSysIn 3-215
CheckReadAsync 3-217
CheckWriteAsync 3-219
CleanQueue 3-221
ClearPath 3-223
CloseAllFiles 3-225
CloseAllFilesLL 3-227
CloseAltMsgFile 3-229
CloseByteStream 3-231
CloseDaFile 3-233
CloseErcFile 3-235
CloseFile 3-237
CloseMsgFile 3-239
ClosePSSession 3-241
CloseRsFile 3-243
CloseRtClock 3-245
CloseServerMsgFile 3-247
CloseTerminal 3-249
CloseVidFilter 3-251
CloseSysCmds 3-253
CompactDateTime 3-255
ConfigCloseFile 3-257

ConfigGetNextToken 3-259
ConfigGetPosition 3-263
ConfigGetRestOfLine 3-265
ConfigOpenFile 3-267
ConfigSetPosition 3-269
ConfigureSpooler 3-271
ControlInterrupt 3-275
ConvertToSys 3-279
CopyFile 3-281
CParams 3-283
Crash 3-285
CreateAlias 3-287
CreateBigPartition 3-289
CreateContext 3-293
CreateDir 3-295
CreateExecScreen 3-297
CreateFile 3-299
CreatePartition 3-303
CreateProcess 3-307
CreateUser 3-311
CSubParams 3-313
CurrentOsVersion 3-315
DCXVersion 3-319
DeallocAliasForServer 3-321
DeallocateRods 3-323
DeallocateSegment 3-324A
DeallocExch 3-325
DeallocHugeMemory 3-326A
DeallocMemoryLL 3-327
DeallocMemorySL 3-329
DeallocRunFile 3-331
DeallocSg 3-333
DeallocUserNumbers 3-335
DebugOp 3-336A
DefineInterLevelStack 3-337
DefineLocalPageMap 3-339
DeinstallQueueManager 3-341
DeinstPSServer 3-343

Delay 3-345
DeleteByteStream 3-347
DeleteDaRecord 3-349
DeleteDir 3-351
DeleteFile 3-353
DeviceInService 3-355
DisableActionFinish 3-357
DisableCluster 3-359
DiscardInputBsC 3-361
DiscardOutputBsC 3-363
DismountVolume 3-365
DmaMapBuffer 3-367
DmaMapBufferFast 3-371
DmaUnmapBuffer 3-375
DoDirectRead 3-377
DoDirectWrite 3-381
Doze 3-385
DrainTerminalOutput 3-387
EnableSwapperOptions 3-389
EnlsAppendChar 3-391
EnlsCase 3-393
EnlsCbToCCols 3-395
EnlsClass 3-397
EnlsDeleteChar 3-401
EnlsDrawBox 3-403
EnlsDrawFormChars 3-407
EnlsDrawLine 3-409
EnlsFieldEdit 3-413
EnlsFieldEditByChar 3-419
EnlsFindC 3-423
EnlsFindRC 3-425
EnlsGetChar 3-427
EnlsGetCharWidth 3-431
EnlsGetPrevChar 3-433
EnlsInsertChar 3-435
EnlsMapCharToStdValue 3-437
EnlsMapStdValueToChar 3-439
EnlsQueryBoxSize 3-441

ErrorExit 3-443
ErrorExitString 3-447
EstablishQueueServer 3-449
Exit 3-451
ExitAndRemove 3-453
ExitListQuery 3-454A
ExitListSet 3-454C
ExpandAreaLL 3-455
ExpandAreaSL 3-457
ExpandDateTime 3-459
ExpandLocalMsg 3-461
ExpandSpec 3-465
FatalError 3-469
FComparePointer 3-471
FieldEdit 3-473
FillBufferAsyncC 3-477
FillBufferC 3-481
FillFrame 3-483
FillFrameRectangle 3-485
FillInRectangle 3-486A
FilterDebugInterrupts 3-487
FlushBufferAsyncC 3-489
FlushBufferC 3-493
FMergedOs 3-495
Format 3-497
FormatDateTime 3-499
FormatTime 3-501
FormatTimeDt 3-503
FormatTimeTm 3-505
FormEdit 3-507
ForwardRequest 3-521
FProcessorSupportsProtectedMode 3-523
FProtectedMode 3-525
FrameBackSpace 3-527
FRmos 3-529
FsCanon 3-531
FSrpUp 3-533
FValidPbCb 3-535

FVSeries 3-536A
GenResString 3-537
GetAltMsg 3-539
GetAltMsgUnexpanded 3-541
GetAltMsgUnexpandedLength 3-543
GetBoardInfo 3-545
GetBsLfa 3-547
GetBsMember 3-548A
GetCanonicalNodeAndVol 3-549
GetClstrGenerationNumber 3-550A
GetClusterId 3-551
GetClusterStatus 3-555
GetCommLineDmaStatus 3-559
GetCoprocessorStatus 3-561
GetCParasOvlyZone 3-563
GetCtosDiskPartition 3-565
GetDateTime 3-567
GetDirInfo 3-569
GetDirStatus 3-571
GetErc 3-573
GetErcLength 3-575
GetFhLongevity 3-577
GetFileErc 3-579
GetFileInfoByName 3-580A
GetFileStatus 3-581
GetFRmosUser 3-587
GetHandleStatus 3-589
GetIBusDevInfo 3-591
GetKbdId 3-595
GetKeyboardId 3-597
GetLocalDaiNumber 3-599
GetMemoryInfo 3-601
GetModuleAddress 3-603
GetModuleId 3-605
GetModuleResourceInfo 3-606A
GetMsg 3-607
GetMsgUnexpanded 3-609
GetMsgUnexpandedLength 3-611

GetNlsDateName 3-613
GetNlsDateTimeTemplate 3-617
GetNlsKeycapText 3-619
GetNlsTable 3-621
GetOvlyStats 3-623
GetPartitionExchange 3-625
GetPartitionHandle 3-627
GetPartitionStatus 3-629
GetPartitionSwapMode 3-633
GetPAscb 3-635
GetPNlsTable 3-637
GetProcInfo 3-639
GetPSCounters 3-643
GetPStructure 3-645
GetQMStatus 3-655
GetRouteTable 3-659
GetRsLfa 3-663
GetScsiInfo 3-665
GetSegmentLength 3-667
GetServerMsg 3-669
GetSlotInfo 3-671
GetStamFileHeader 3-675
GetStandardErcMsg 3-677
GetSysCmdInfo 3-679
GetUcb 3-681
GetUserFileEntry 3-683
GetUserLocation 3-687
GetUserNumber 3-689
GetUserStatus 3-691
GetVhb 3-695
GetWsUserName 3-697
HeapAlloc 3-699
HeapFree 3-701
HeapInit 3-703
InitAltMsgFile 3-705
InitCharMap 3-707
InitCommLine 3-709
InitErcFile 3-719

InitLargeOverlays 3-721
InitLocalPageMap 3-723
InitMsgFile 3-725
InitOverlays 3-727
InitSysCmds 3-729
InitVidFrame 3-731
InstallNet 3-735
KeyboardProfile 3-737
KillProcess 3-739
LaFromP 3-740A
LaFrom 3-740C
LibFree 3-740E
LibGetHandle 3-740G
LibGetInfo 3-740I
LibGetProcInfo 3-740K
LibLoad 3-740M
LoadBackgroundPalette 3-741
LoadColorStyleRam 3-743
LoadFontRam 3-745
LoadInteractiveTask 3-749
LoadPrimaryTask 3-751
LoadRunFile 3-755
LoadTask 3-757
LockIn 3-761
LockInContext 3-763
LockOut 3-765
LockVideo 3-767
LockVideoForModify 3-769
LockXbis 3-771
LogMsgIn 3-773
LogRespond 3-775
LogRequest 3-777
LookUpField 3-779
LookUpNumber 3-781
LookUpReset 3-783
LookUpString 3-785
LookUpValue 3-786A
MakePermanent 3-787

MakePermanentP 3-789
MakeRecentlyUsed 3-791
MapBusAddress 3-793
MapCsIOvly 3-797
MapIOvlyCs 3-799
MapPhysicalAddress 3-801
MapPStubPProc 3-805
MapXBusWindow 3-807
MapXBusWindowLarge 3-811
MarkKeyedQueueEntry 3-815
MarkNextQueueEntry 3-819
McrVersion 3-821
MediateIntHandler 3-823
MenuEdit 3-825
Mode3DmaReload 3-833
MountVolume 3-835
MouseVersion 3-837
MoveFrameRectangle 3-839
MoveOverlays 3-841
NameAllocClass 3-842A
NameQuery 3-842C
NameRegister 3-842E
NameRemove 3-842G
NewProcess 3-843
NlsCase 3-847
NlsClass 3-849
NlsCollate 3-851
NlsFormatDateTime 3-855
NlsGetYesNoStrings 3-859
NlsGetYesNoStringSize 3-861
NlsNumberAndCurrency 3-863
NlsParseTime 3-865
NlsSpecialCharacters 3-867
NlsStdFormatDateTime 3-869
NlsULCmpB 3-873
NlsVerifySignatures 3-875
NlsYesNoOrBlank 3-877
NlsYesOrNo 3-879

NPrint 3-881
ObtainAccessInfo 3-883
ObtainUserAccessInfo 3-887
OldReadHardId 3-891
OldWriteHardId 3-893
OpenByteStream 3-895
OpenByteStreamC 3-897
OpenDaFile 3-899
OpenFile 3-901
OpenFileLL 3-905
OpenNlsFile 3-909
OpenPSLogSession 3-911
OpenPSStatSession 3-913
OpenRsFile 3-915
OpenRtClock 3-917
OpenServerMsgFile 3-919
OpenTerminal 3-921
OpenUserFile 3-925
OpenVidFilter 3-927
OsVersion 3-929
OutputBytesWithWrap 3-933
OutputQuad 3-935
OutputToVid0 3-937
OutputWord 3-939
PaFromP 3-941
PaFromSn 3-943
ParseFileSpec 3-945
ParseSpecForDir 3-951
ParseSpecForFile 3-955
ParseSpecForNode 3-957
ParseSpecForPassword 3-961
ParseSpecForVol 3-965
ParseTime 3-969
PDAssignMouse 3-971
PDGetCursorPos 3-973
PDGetCursorPosNSC 3-975
PDInitialize 3-977
PDLoadCursor 3-979

PDLoadSystemCursor 3-985
PDQueryControls 3-987
PDQuerySystemControls 3-989
PDReadCurrentCursor 3-991
PDReadIconFile 3-995
PDSetCharMapVirtualCoordinates 3-997
PDSetControls 3-999
PDSetCursorDisplay 3-1001
PDSetCursorPos 3-1003
PDSetCursorPosNSC 3-1005
PDSetCursorType 3-1007
PDSetMotionRectangle 3-1009
PDSetMotionRectangleNSC 3-1011
PDSetSystemControls 3-1013
PDSetTracking 3-1015
PDSetVirtualCoordinates 3-1017
PDTranslateNSCToVC 3-1019
PDTranslateVCToNSC 3-1021
PosFrameCursor 3-1023
PostKbdTable 3-1025
PrintAltMsg 3-1029
PrintErc 3-1031
PrintFileClose 3-1033
PrintFileOpen 3-1035
PrintFileStatus 3-1037
PrintMsg 3-1039
ProgramColorMapper 3-1041
PSCloseSession 3-1045
PSDeinstServer 3-1047
PSend 3-1049
PSGetCounters 3-1051
PSOpenLogSession 3-1055
PSOpenStatSession 3-1059
PSReadLog 3-1063
PSResetCounters 3-1067
PurgeMcr 3-1069
PutBackByte 3-1071
PutByte 3-1073

PutChar 3-1075
PutCharsAndAttrs 3-1077
PutFrameAttrs 3-1079
PutFrameChars 3-1081
PutFrameCharsAndAttrs 3-1083
PutPointer 3-1085
PutQuad 3-1087
PutWord 3-1089
QueryBigMemAvail 3-1091
QueryBoardInfo 3-1093
QueryBounds 3-1097
QueryCharsAndAttrs 3-1099
QueryCoprocessor 3-1101
QueryCursor 3-1103
QueryDaLastRecord 3-1105
QueryDaRecordStatus 3-1107
QueryDcb 3-1109
QueryDefaultRespExch 3-1111
QueryDeviceName 3-1113
QueryDeviceNames 3-1117
QueryDiskGeometry 3-1119
QueryExitRunFile 3-1125
QueryFrameAttrs 3-1127
QueryFrameBounds 3-1129
QueryFrameChar 3-1131
QueryFrameCharsAndAttrs 3-1133
QueryFrameCursor 3-1135
QueryFrameString 3-1137
QueryIOOwner 3-1139
QueryKbdLeds 3-1143
QueryKbdState 3-1145
QueryLdtR 3-1147
QueryMail 3-1149
QueryMemAvail 3-1151
QueryModulePosition 3-1153
QueryNodeName 3-1155
QueryPagingStatistics 3-1156A
QueryProcessInfo 3-1157

QueryProcessNumber 3-1161
QueryRequestInfo 3-1163
QueryTrapHandler 3-1165
QueryUserLocation 3-1169
QueryVidBs 3-1171
QueryVideo 3-1173
QueryVidHdw 3-1175
QueryWsNum 3-1183
QueryZoomBoxPosition 3-1185
QueryZoomBoxSize 3-1187
QueueMgrVersion 3-1189
Read 3-1191
ReadActionCode 3-1193
ReadActionKbd 3-1195
ReadAsync 3-1197
ReadBsRecord 3-1199
ReadByte 3-1201
ReadBytes 3-1203
ReadByteStreamParameterC 3-1205
ReadCommLineStatus 3-1207
ReadDaFragment 3-1209
ReadDaRecord 3-1211
ReadDirSector 3-1213
ReadHardId 3-1215
ReadInputEvent 3-1217
ReadInputEventNSC 3-1223
ReadKbd 3-1225
ReadKbdDataDirect 3-1227
ReadKbdDirect 3-1233
ReadKbdInfo 3-1237
ReadKbdStatus 3-1243
ReadKeyedQueueEntry 3-1247
ReadKeySwitch 3-1251
ReadMcr 3-1253
ReadNextQueueEntry 3-1257
ReadOsKbdTable 3-1259
ReadPSLog 3-1263
ReadRemote 3-1265

ReadRsRecord 3-1267
ReadStatusC 3-1269
ReadTerminal 3-1271
ReadToNextField 3-1275
ReallocHugeMemory 3-1276A
ReceiveCommLineDma 3-1277
ReinitLargeOverlays 3-1279
ReinitOverlays 3-1281
ReinitStubs 3-1283
ReleaseByteStream 3-1285
ReleaseByteStreamC 3-1287
ReleasePermanence 3-1289
ReleaseRsFile 3-1291
RemakeAliasForServer 3-1293
RemakeFh 3-1295
RemapBusAddress 3-1299
RemoteBoot 3-1303
RemoveFfsBrackets 3-1307
RemoveKeyedQueueEntry 3-1309
RemoveMarkedQueueEntry 3-1311
RemovePartition 3-1313
RemoveQueue 3-1315
RenameByteStream 3-1317
RenameFile 3-1319
ReopenFile 3-1321
Request 3-1323
RequestDirect 3-1325
RequestRemote 3-1327
RescheduleMarkedQueueEntry 3-1329
RescheduleProcess 3-1331
ReserveBusAddress 3-1333
ReservePartitionMemory 3-1335
ResetCommLine 3-1339
ResetDeviceHandler 3-1341
ResetFrame 3-1343
ResetIBusHandler 3-1345
ResetMemoryLL 3-1347
ResetPSCounters 3-1349

ResetTimerInt 3-1351
ResetTrapHandler 3-1353
ResetVideo 3-1355
ResetVideoGraphics 3-1359
ResetXBusMIsr 3-1365
ResizeIoMap 3-1367
ResizeSegment 3-1368A
Respond 3-1369
ResumeContext 3-1371
ReuseAlias 3-1373
ReuseAliasLarge 3-1375
RewriteMarkedQueueEntry 3-1377
RgParam 3-1379
RgParamInit 3-1381
RgParamSetEltNext 3-1383
RgParamSetListStart 3-1385
RgParamSetSimple 3-1387
RkvsVersion 3-1389
RsrcCopyFromFile 3-1390A
RsrcCopyFromRsrcSet 3-1390E
RsrcCopyRestRunFile 3-1390G
RsrcDelete 3-1390I
RsrcEndSetAccess 3-1390K
RsrcGetAllSetTypeInfo 3-1390M
RsrcGetCountAllSetTypes 3-1390O
RsrcGetCountSetType 3-1390Q
RsrcGetDesc 3-1390S
RsrcGetSetTypeInfo 3-1390U
RsrcInitSetAccess 3-1390W
RsrcSessionEnd 3-1390Z1
RsrcSessionInit 3-1390Z3
SbPrint 3-1391
ScanToGoodRsRecord 3-1393
ScreenPrintVersion 3-1395
ScrollFrame 3-1397
ScrubFile 3-1399
ScsiCdbDataIn 3-1401
ScsiCdbDataInAsync 3-1405

ScsiCdbDataOut 3-1409
ScsiCdbDataOutAsync 3-1413
ScsiClosePath 3-1417
ScsiManagerNameQuery 3-1419
ScsiOpenPath 3-1421
ScsiQueryInfo 3-1427
ScsiQueryPathParameters 3-1433
ScsiRequestSense 3-1437
ScsiReset 3-1441
ScsiSetPathParameters 3-1443
ScsiTargetCdbCheck 3-1447
ScsiTargetCdbWait 3-1449
ScsiTargetDataReceive 3-1451
ScsiTargetDataReceiveAsync 3-1455
ScsiTargetDataTransmit 3-1457
ScsiTargetDataTransmitAsync 3-1461
ScsiTargetOperationsAbort 3-1463
ScsiWaitCdbAsync 3-1465
ScsiWaitTargetDataAsync 3-1467
SemClear 3-1468A
SemClearCritical 3-1468C
SemClearProcessLocks 3-1468E
SemClose 3-1468G
SemEnumerate 3-1468I
SemEnumerateWaiting 3-1468K
SemLock 3-1468O
SemLockCritical 3-1468Q
SemMuxWait 3-1468S
SeNotify 3-1468W
SemOpen 3-1468Y
SemQuery 3-1468Z3
SemQueryProcessLock 3-1468Z9
SemSet 3-1468Z11
SemWait 3-1468Z13
SendBreakC 3-1471
SeqAccessCheckpoint 3-1473
SeqAccessClose 3-1475
SeqAccessControl 3-1477

SeqAccessDiscardBufferData 3-1485
SeqAccessModeQuery 3-1487
SeqAccessModeSet 3-1491
SeqAccessOpen 3-1495
SeqAccessRead 3-1505
SeqAccessRecoverBufferData 3-1509
SeqAccessStatus 3-1513
SeqAccessVersion 3-1517
SeqAccessWrite 3-1521
SerialNumberQuery 3-1525
SerialNumberOldOsQuery 3-1527
ServeRq 3-1529
Set386TrapHandler 3-1531
SetAlphaColorDefault 3-1533
SetBsLfa 3-1535
SetDaBufferMode 3-1537
SetDateTime 3-1539
SetDateTimeMode 3-1541
SetDefault386TrapHandler 3-1542A
SetDefaultTrapHandler 3-1543
SetDeltaPriority 3-1545
SetDeviceHandler 3-1547
SetDevParams 3-1549
SetDirStatus 3-1553
SetDiskGeometry 3-1557
SetDispMsw287 3-1561
SetExitRunFile 3-1563
SetFhLongevity 3-1565
SetField 3-1567
SetFieldNumber 3-1569
SetFileStatus 3-1571
SetIBusHandler 3-1577
SetImageMode 3-1581
SetImageModeC 3-1583
SetIntHandler 3-1585
SetIOOwner 3-1589
SetKbdLed 3-1593
SetKbdUnencodedMode 3-1597

SetKeyboardOptions 3-1599
SetLdtRDs 3-1603
SetLPIsr 3-1605
SetModuleId 3-1606A
SetMsgRet 3-1607
SetNode 3-1609
SetPartitionExchange 3-1611
SetPartitionLock 3-1613
SetPartitionName 3-1615
SetPartitionSwapMode 3-1617
SetPath 3-1621
SetPrefix 3-1623
SetPStructure 3-1625
SetRsLfa 3-1629
SetScreenVidAttr 3-1631
SetSegmentAccess 3-1633
SetStyleRam 3-1637
SetStyleRamEntry 3-1639
SetSwapDisable 3-1641
SetSysInMode 3-1643
SetTerminal 3-1647
SetTimerInt 3-1653
SetTrapHandler 3-1655
SetUpMailNotification 3-1657
SetUserFileEntry 3-1659
SetVideoTimeout 3-1663
SetWsUserName 3-1665
SetXBusMIsr 3-1667
SgFromSa 3-1671
ShortDelay 3-1675
ShrinkAreaLL 3-1677
ShrinkAreaSL 3-1679
ShrinkPartition 3-1681
SignOnLog 3-1683
SnFromSr 3-1687
SpoolerPassword 3-1689
SpoolerVersion 3-1691
SrFromSn 3-1693

StatisticsVersion 3-1695
StringsEqual 3-1697
SuspendProcess 3-1699
SuspendUser 3-1701
SwapContextUser 3-1703
SwapInContext 3-1705
SwappingRequests 3-1707
SwapXBusEar 3-1709
SystemCommonCheck 3-1711
SystemCommonConnect 3-1713
SystemCommonInstall 3-1715
SystemCommonQuery 3-1719
TerminateAllOtherContexts 3-1721
TerminateContext 3-1723
TerminateContextUser 3-1725
TerminatePartitionTasks 3-1727
TerminateQueueServer 3-1729
TerminationRequests 3-1731
TextEdit 3-1733
TransmitCommLineDma 3-1737
TruncateDaFile 3-1739
TsConnect 3-1741
TsDataChangeParams 3-1745
TsDataCheckpoint 3-1747
TsDataCloseLine 3-1749
TsDataOpenLine 3-1751
TsDataRead 3-1755
TsDataRetrieveParams 3-1757
TsDataUnacceptCall 3-1759
TsDataWrite 3-1761
TsDeinstall 3-1763
TsDial 3-1765
TsDoFunction 3-1767
TsGetStatus 3-1771
TsHold 3-1773
TsLoadCallProgressTones 3-1775
TsOffHook 3-1777
TsOnHook 3-1779

TsQueryConfigParams 3-1781
TsReadTouchTone 3-1783
TsRing 3-1787
TsSetConfigParams 3-1789
TsVersion 3-1791
TsVoiceConnect 3-1793
TsVoicePlaybackFromFile 3-1795
TsVoiceRecordToFile 3-1799
TsVoiceStop 3-1803
TurnOffMailNotification 3-1805
ULCmpB 3-1807
UndeleteFile 3-1809
UnlockInContext 3-1811
UnlockVideo 3-1813
UnlockVideoForModify 3-1815
UnlockXbis 3-1817
UnmapBusAddress 3-1819
UnmapPhysicalAddress 3-1821
UnmarkQueueEntry 3-1823
UnsuspendProcess 3-1825
UnsuspendUser 3-1827
UnzoomBox 3-1829
UpdateOverlayLru 3-1831
VacatePartition 3-1833
Wait 3-1835
WaitLong 3-1837
WhereTerminalBuffer 3-1839
WildCardClose 3-1843
WildCardInit 3-1845
WildCardMatch 3-1847
WildCardNext 3-1849
Write 3-1851
WriteAsync 3-1853
WriteBsRecord 3-1855
WriteByte 3-1857
WriteByteStreamParameterC 3-1859
WriteCommLineStatus 3-1861
WriteDaFragment 3-1863

WriteDaRecord 3-1865
WriteHardId 3-1867
WriteIBusDevice 3-1869
WriteIBusEvent 3-1871
WriteKbdBuffer 3-1875
WriteLog 3-1879
WriteRemote 3-1881
WriteRsRecord 3-1883
WriteStatusC 3-1885
XbifVersion 3-1887
ZoomBox 3-1889
ZPrint 3-1893

4 System Structures
Chapter Organization 4-1
System Structures Table of Contents 4-2
Allowed States Table 4-5
Application System Control Block 4-9
Batch Control Block 4-17
Cache Entry Descriptor 4-23
Cache Pool Descriptor 4-27
Chords Table 4-31
Command Masks Table 4-37
Communications Configuration Descriptor 4-41
Communications Status Buffer 4-47
Conditions Table 4-67
Control Chords Table 4-71
CPU Description Table 4-75
CPU Description Table (pre-3.0) 4-87
Data Control Structure 4-95
Decoding Offsets Table 4-101
Decoding Table 4-105
Device Control Block 4-109
Device Control Block (pre-3.0) 4-117
Diacritic Masks Table 4-127
Diacritics Table 4-131
Directory Entry in Master File Directory Block 4-135
Emulation Data Block Header 4-139

Emulation LEDs Table 4-145
Emulation Table 4-149
Expanded Date/Time Format 4-153
Extended Partition Descriptor 4-157
Extended Process Control Block 4-165
File Entry in a Directory Block 4-169
File Header Block 4-173
Frame Descriptor 4-183
High Sierra Directory Record 4-189
ISO Directory Record 4-191
Log File Record Format 4-193
Low Memory Allocation 4-213
Master File Directory Block 4-219
Multibyte String Masks Table 4-222A
Multibyte String Offsets Table 4-222E
Multibyte Strings Table 4-222I
Operating System Flags 4-223
Partition Configuration Block 4-229
Partition Descriptor 4-233
Partition Information Block 4-239
Partition Keyboard Information Structure 4-243
Performance Statistics Structure 4-251
Port Structure 4-257
Process Control Block 4-271
Queue Entry Header 4-275
Queue File Header 4-279
Queue Status Block 4-285
Repeat Attributes Table 4-289
Resource Descriptor 4-292A
Special Keys Table 4-293
Spooler Control Queue Entry 4-299
Spooler Scheduling Queue Entry 4-303
Spooler Status Queue Entry 4-309
Standard File Header Format 4-315
Standard Record Header Format 4-319
Standard Record Trailer Format 4-323
String Masks Table 4-327
String Offsets Table 4-331

Strings Table 4-335
System Configuration Block 4-339
System Date/Time Structure 4-347
Table Descriptor Array 4-351
Telephone Service Configuration Block 4-355
Telephone Service Configuration File Format 4-361
Telephone Status Structure 4-365
Terminal Output Buffer 4-375
Timer Pseudointerrupt Block 4-379
Timer Request Block Format 4-383
Translation Data Block Header 4-387
Translation Table 4-393
User Control Block 4-397
Variable Length Parameter Block 4-401
Video Control Block 4-405
Voice Control Structure 4-413
Voice File Header 4-419
Voice File Record 4-425
Voice Processor Control Block 4-429
Volume Home Block 4-437
Volume Home Block (pre-3.0) 4-451

Appendix A
Status Codes A-1

Appendix B
Standard Character Set B-1

Appendix C
Keyboard Codes C-1

Appendix D
Request Codes in Numeric Sequence D-1

Appendix E
NLS Templates E-1

Appendix F
Message File Macro Definitions F-1

Appendix G
X-Bus and I-Bus Module IDs G-1

Appendix H
Byte Stream Escape Sequences H-1

Appendix I
System Common Procedure Parameter Syntax I-1

Appendix J
Operation Data J-1

Appendix K
Resource Types K-1

List of Figures

2-1. Operations by Function 2-2
3-1. Form Showing a String Response 3-508
3-2. Form Showing Yes or No Choice Responses 3-508
3-3. Menu With One Item Selected 3-825
3-4. Menu With Three Items Selected 3-826
3-5. User-Defined Graphics Cursor 3-981
3-6. Six Rows of Sample Graphics Cursor Loaded into Screen Memory 3-981
3-7. Pascal Code Defining the Left-Arrow Cursor 3-982

List of Tables

ATM-1 Common Prefixes xli
ATM-2 Common Roots ..... xliB
B-1. Standard Character Set B-2
B-2. Graphic Representation of the Standard Character Set B-13
B-3. Nationalized Video Display Characters B-15
C-1. Keyboard Codes Generated by an Unencoded K1 Keyboard C-2
C-2. Keyboard Codes: Key Differences for K2, K3, K5, and SG101-K keyboards C-6
C-3. Keyboard Codes Generated by an Unencoded SG102-K Keyboard C-7
E-1. NLS Templates E-2
G-1. Module IDs G-5
H-2. Escape Sequence Redirecting a Video Byte Stream .. H-1
H-3. Escape Sequence Controlling Character Attributes .. H-2
H-4. Escape Sequence Controlling Pause H-4
H-5. Escape Sequence Controlling Scrolling and Cursor Positioning H-5
H-6. Escape Sequence Performing Miscellaneous Functions H-6
H-7. Video Byte Stream Interpretation of Special Characters H-8

H-8. Submit File Escape Sequences H-9
J-1. Kernel Primitives J-2
J-2. Requests J-3
J-3. System Common Procedures J-17
J-4. Standard Operating System Library Procedures J-21
J-5. Mouse Library Procedures J-30
J-6. Asynchronous System Service Library Procedures... J-31
K-1. Resource Type Codes K-1

Purpose and Scope

This manual is the programmer's reference to CTOS operating systems. Specifically, it describes the programmatic interfaces to the operating systems currently supported. These are

- CTOS III, Version 1.0
- CTOS II, Version 3.4
- CTOS I, Version 3.4
- CTOS/XE, Version 3.4

Collectively, these operating systems are called *CTOS* in this manual. The only time the text calls out a specific operating system name is when a feature unique to that system is being described. In most cases, you can determine whether or not a feature is supported on your system; if the text does not indicate limited support, the feature is part of the common CTOS software base. For more information, see the *CTOS Operating System Concepts Manual*.

In the text of this manual, the Unisys XE-520 and XE-530 Shared Resource Processors and the Convergent Shared Resource Processor™ are all referred to collectively as a shared resource processor.

The programmatic interfaces in Chapter 3 in this manual are detailed descriptions of the routines in the CTOS System Image and in the standard operating system libraries, CTOS.lib, CTOSToolkit.lib, and Enls.lib. CTOS.lib, CTOSToolkit.lib, and Enls.lib, hereafter called the *standard operating system libraries*, are distributed with the Standard Software Development Utilities.

In addition to the procedures in the System Image and the standard operating system libraries, Chapter 3 describes procedures for those additional system services included with the Standard Software Development Utilities that can extend the functionality of the operating system when they are installed. In this manual, these services are called extended system services. They include the following:

- Asynchronous System Service
- CD-ROM Service
- Command Access Service
- Mouse Services
- Performance Statistics System Service
- Queue Manager
- Sequential Access Service
- Spooler
- Voice/Data Services (including the new Audio Service)

Mouse Services and the Asynchronous System Service consist of library procedures in Mouse.lib and Async.lib, respectively. The Queue Manager, Spooler, Voice/Data Services, CD-ROM Service, Sequential Access Service, and the Performance Statistics System Service consist of dynamically installable requests. These services are supported on all CTOS systems.

Two other manuals are very closely related to the *CTOS Procedural Interface Reference Manual*. These manuals are the *CTOS Operating System Concepts Manual* and the *CTOS Programming Guide*. The *CTOS Operating System Concepts Manual* guides you through an overview of each CTOS subsystem. In addition it contains a glossary of operating system terms. The *CTOS Programming Guide* addresses the needs of applications and systems programmers by providing detailed program examples. It is a two volume set that shows how to use the extended system services programmatically as well as how to use other system and library routines in applications you write.

Restricted Use Only

Some of the operations in this manual are labeled as restricted use only. These operations are not intended for widespread use. Characteristics of these operations are

- The operation may not be supported in a future release.
- The interface (parameters, returned data, error codes, and so forth) may change in a future release.
- Use of the operation may be restricted to certain application programs or system services.
- Reported defects or problems will not necessarily be repaired or evaluated.
- The operation may not be fully documented and may have undesirable side affects if used.

Internal Use Only

Some CTOS operations are labeled as internal use only. Distribution of Volume 5 of this manual, which includes complete documentation for these operations, is restricted. However, some internal-use only features are included here. If included, they are documented only to facilitate system behavior analysis. Internal-use features have all of the restrictions that apply to restricted-use operations, described above.

Organization of This Manual

This manual is organized as follows:

- Chapter 1 introduces you to the operations that are each described in detail in Chapter 3.
- Chapter 2 groups the operations by their common function. (See Figure 2-1.) The group names are set up in such a way that, if you need more background on how a particular operation relates to others and to the operating system as a whole, you can find a chapter corresponding to the operation's group name in the *CTOS Operating System Concepts Manual*.
- Chapter 3 alphabetically presents all the operations. For quick lookup, the operation name is shown at the top of each page.
- Chapter 4 presents the system structures. Within the chapter, the system structures are presented alphabetically. For each structure, there is a brief introduction explaining the significance of the structure, how to access it, and where to go for more information. Following the introduction, the format of the structure is shown.
- The appendixes contain other useful programming information, including

  error codes

  standard character set

  keyboard codes

  request codes

  Native Language Support (NLS) templates

  message file macro definitions

  X-Bus and I-Bus module IDs

  byte stream escape sequences

  parameter syntax for system common procedures

  internal version numbers of each operation

  resource descriptors

NOTE: To determine if an operation is supported on your system, you can consult Appendix J, "Operating System Data." The appendix indicates the internal version number of each operation described in the System Image and the standard operating system libraries. To achieve the greatest flexibility in creating portable programs, you should always link your programs with the appropriate version of Standard Software. (See your release documentation for details on the version you should use.)

What is New in This Manual

This is a revised edition of the CTOS II 3.3 base manual. Update Notice 1 documents additions and changes to the CTOS III 1.0, CTOS II 3.4, CTOS I 3.4, and CTOS/XE 3.4 operating systems. This section first lists the changes that were included in the CTOS II 3.3 edition, then it lists the changes for the Update Notice.

What was New for the CTOS II 3.3 Edition

The *CTOS Procedural Interface Reference Manual* differs from the previous Unisys and Convergent operating system manuals in a number of significant ways. The differences are enumerated in detail below. This manual

1. Describes all the new system operations (system-common procedures and requests) introduced with the CTOS I 3.3 and CTOS II 3.3 operating systems. Alphabetically, these operations are

   AccessVersion
   DmaMapBuffer
   DmaMapBufferFast
   DmaUnmapBuffer
   GetKeyboardId
   GetModuleAddress
   MapPhysicalAddress
   PostKbdTable
   QueryIOOwner
   QueryProcessInfo
   QueryTrapHandler
   ReadKbdInfo
   ReadOsKbdTable
   ResetDeviceHandler
   ResetTrapHandler
   ResizeIoMap
   ScreenPrintVersion
   SetIOOwner
   SerialNumberQuery
   SetKeyboardOptions
   UnmapPhysicalAddress
   WriteKbdBuffer

2. Describes all the new library procedures introduced with the Standard Software Development Utilities, Version 12.1. These are

ConfigCloseFile
ConfigGetNextToken
ConfigGetPosition
ConfigGetRestOfLine
ConfigOpenFile
ConfigSetPosition
EnlsAppendChar
EnlsCbToCCols
EnlsCase
EnlsClass
EnlsDeleteChar
EnlsDrawBox
EnlsDrawFormChars
EnlsDrawLine
EnlsFieldEdit
EnlsFieldEditByChar
EnlsFindC
EnlsFindRC
EnlsGetChar
EnlsGetCharWidth
EnlsGetPrevChar
EnlsInsertChar
EnlsMapCharToStdValue
EnlsMapStdValueToChar
EnlsQueryBoxSize
FMergedOs
GetCtosDiskPartition
GetNlsDateTimeTemplate
KeyboardProfile
ObtainAccessInfo
ObtainUserAccessInfo
QueryZoomBoxSize
SerialNumberOldOsQuery
WildCardClose

3. Describes all the new requests served by extended system services. These are

Voice/Data Services Requests (Including Audio Service Requests)

AsGetVolume
AsSetVolume
TsLoadCallProgressTones

CD-ROM Service Requests

CdAbsoluteRead
CdAudioCtl
CdControl
CdClose
CdDirectoryList
CdGetDirEntry
CdGetVolumeInfo
CdOpen
CdRead
CdSearchClose
CdSearchFirst
CdSearchNext
CdServiceControl
CdVerifyPath
CdVersion

Sequential Access Service Requests

SeqAccessCheckpoint
SeqAccessClose
SeqAccessControl
SeqAccessDiscardBufferData
SeqAccessModeQuery
SeqAccessModeSet
SeqAccessOpen
SeqAccessRead
SeqAccessRecoverBufferData
SeqAccessStatus
SeqAccessVersion
SeqAccessWrite

4. Describes all new structures shared by two or more operations. These structures are

CD-ROM Structures (extended system service)

High Sierra Directory Record
ISO Directory Record

Keyboard Structures

Allowed States Table
Chords Table
Command Masks Table
Conditions Table
Control Chords Table
Decoding Offsets Table
Decoding Table
Diacritic Masks Table
Diacritics Table
Emulation Data Block Header
Emulation LEDs Table
Emulation Table
Multibyte String Masks Table
Multibyte String Offsets Table
Multibyte Strings Table
Partition Keyboard Information Structure
Repeat Attributes Table
Special Keys Table
Table Descriptor Array
Translation Data Block Header
Translation Table

Partition Structures

Partition Information Block

5. Does not document the following superceded operations:

AllocMoverSegment	QicSyncReadQicHeader
CloseTape	ReadTapeRecords
DeallocMoverSegment	TapeOperation
GetSerialNumber	TapeStatus
GetSerialNumberOldOs	TapeVersion
LoadKbdXlat	WriteTapeRecords
LoadKbdXlatFh	WriteLed
MovbMoverSegment	WriteQicHeader
OpenTape	

In addition, some operations and system structures used only by the operating system have been classified as internal-use only and have been placed in a separate volume that is available only to Unisys and OEMs. Those operations are still listed in Appendix D and Appendix J.

What is New in this Update Notice

Update Notice 1 describes all the new operations introduced in the CTOS III 1.0 operating system. It also fixes various errata to the 3.3 edition base manual.

CTOS III 1.0 runs on all 80386 and 80486 CTOS workstations, including a new style workstation. This Update Notice uses the phrase EISA/ISA-bus workstation to describe the new style workstation, which consists of a base unit with either an EISA board or an ISA board, a hard disk, a floppy disk drive, a video controller, as well as other hardware components. The SuperGen Series 3000 is an EISA/ISA-bus workstation.

The new operations in Update Notice 1 are listed below.

1. Cluster Management

   GetClstrGenerationNumber

2. Demand Paging

   QueryPagingStatistics

3. Dynamic Link Libraries

   ExitListQuery
   ExitListSet
   LibFree
   LibGetHandle
   LibGetInfo
   LibGetProcInfo
   Libload

4. File Management, File Attributes

   GetFileInfoByName

5. Memory Management

   AllocateSegment
   AllocCommDmaBuffer
   AllocHugeMemory
   DeallocateSegment
   DeallocHugeMemory
   ReallocHugeMemory
   ResizeSegment

6. Memory Management, Selector Management

   LaFromP
   LaFromSn

7. Name Management

   NameAllocClass
   NameRegister
   NameQuery
   NameRemove

8. Operating System Management

   FVSeries

9. Parsing Configuration Files, Parsing Standard Configuration Files

   LookUpValue

10. Resource Management, Accessing Resources in Disk Files

RsrcCopyFromFile
RsrcCopyFromRsrcSet
RsrcCopyRestRunFile
RsrcDelete
RsrcEndSetAccess
RsrcGetAllSetTypeInfo
RsrcGetCountAllSetTypes
RsrcGetCountSetType
RsrcGetDesc
RsrcGetSetTypeInfo
RsrcInitSetAccess
RsrcSessionEnd
RsrcSessionInit

11. Resource Management, Accessing Resources in Memory

GetModuleResourceInfo

12. Semaphore Management

SemClear
SemClearCritical
SemClearProcessLocks
SemClose
SemEnumerate
SemEnumerateWaiting
SemLock
SemLockCritical
SemMuxWait
SemNotify
SemOpen
SemQuery
SemQueryProcessLock
SemSet
SemWait

13. Sequential Access Method

GetBsMember

13. X-Bus Management

    SetModuleId

14. Video, VAM Management

    FillInRectangle

15. The following superceded operations are not documented:

    DmaMapBufferFast
    PaFromP
    PaFromSn

In addition, some operations and system structures used only by the operating system have been classified as internal-use only and have been placed in a separate volume that is available only to Unisys and OEMs. Those operations are still listed in Appendix D and Appendix J.

Related Documentation

This manual is one of a related manual set that documents the Unisys family of information processing systems. For a description of each manual in the set and for order information, see your sales representative or the *Unisys Customer Product Information Catalog*.

The manuals in the operating system set are listed below. Programming documentation exists for other separately orderable products. For more information, see the "Related Documentation" sections in the product manuals.

The following manuals are distributed with the operating system:

- *CTOS Basic Asynchronous Terminal Emulator User's Guide*
- *CTOS Editor User's Guide*
- *CTOS Executive Reference Manual*
- *CTOS Executive User's Guide*
- *CTOS Operating System Concepts Manual*
- *CTOS Status Codes Reference Manual*
- *CTOS System Administration Guide*

The following manual is overpacked with the operating system:

- *CTOS Batch Manager II Installation, Configuration, and Programming Guide*

The following manuals are distributed with the Development Utilities and are also available separately:

- *CTOS Debugger User's Guide*
- *CTOS Programming Utilities Reference Manual: Building Applications*
- *CTOS Programming Utilities Reference Manual: Customization*
- *CTOS Programming Utilities Reference Manual: Assembler*
- *CTOS/Open Programming Practices and Standards*
- *CTOS Programming Guide*

The following manuals are published by Intel Corporation:

- *iAPX 286 Programmer's Reference Manual*
- *80386 Programmer's Reference Manual*
- *i486 Programmer's Reference Manual*

The following manuals are available with separately orderable products:

- *CTOS Indexed Sequential Access Method (ISAM II) Programming Reference Manual*
- *Graphics Programming Guide*
- *Generic Print System Administration Guide*
- *Generic Print System Programming Guide*
- *CTOS OFIS Spreadsheet Reference Manual*

The following manuals are published by other outside sources:

- *Small Computer Systems Interface (SCSI-1), ANSI X3.131-1986* (The ANSI authorized standard for SCSI implementation, available through ANSI)
- *Enhanced Small Computer Systems Interface (SCSI-2), ANSI X 3T9.2/86-109* (Revision 5, or above, available through ANSI)

Naming Conventions Used in This Manual

Memory Address

Memory address refers to the logical memory address. (For details on addressing memory, see "Using CTOS Operations" in the *CTOS Operating System Concepts Manual.*)

Variable Names

The name of the variable implies some of its characteristics. Parameters used in procedure definitions and fields of request blocks and other data structures are named according to this convention.

A variable name is composed of up to three parts: a prefix, a root, and a suffix.

Prefixes

The prefix identifies the data type of the variable. Common prefixes and the number of bytes required for each are shown in Table ATM-1. Note that a flag passed to the operating system or returned by an operating system procedure is interpreted as meaning TRUE if its value is 0FFh and FALSE if its value is 0. Even if you are using a high-level language that uses a different value for TRUE or FALSE, you must use these values to mean TRUE and FALSE in operating system calls.

Table ATM-1. Common Prefixes

Prefix	Bytes Required	Description
b	1	Byte (character or unsigned integer).
c	2	Count (unsigned integer).
cb	2	Count of bytes (in a string of bytes).
f	1	Flag (TRUE = 0FFh and FALSE = 0).
i	2	Index (unsigned integer).
l	varies	literal (a constant).
n	2	Number (unsigned integer) (same as c).
o	2	Offset from the segment base address.
p	4	Logical memory address (pointer) consisting of a segment address and an offset.
pb	4	Logical memory address of a string of bytes.
q	4	Quad (unsigned integer).
rg	varies	Array. Usually used with another prefix, for example, the prefix *rgb* identifies an array of bytes.
mp	varies	Map. A one-to-one correspondence between two variables. Usually used with other variables being mapped, for example, *mpcParcRq* maps the count of paragraphs to the count of requests of that size
s	2	Size in bytes (unsigned integer).
sb	varies	String. An array of bytes where first byte is the size of the string.
sz	varies	String. An array of bytes with a null terminating byte.
w	2	word

Roots

The root can be a unique name for a variable (such as *clientA*), it can be a general name such as the examples shown in Table ATM-2, or it can be a combination of a unique name and a general name (such as *exchClientA*). Common roots and the number of bytes required for each are shown in Table ATM-2.

Table ATM-2. Common Roots

Root	Bytes Required	Description
erc	2	Error code.
exch	2	Exchange.
fh	2	File handle.
fhb	*	File Header Block (FHB). (See "System Structures" in the *CTOS Procedural Interface Reference Manual* for the size of the FHB system structure.)
lfa	4	Logical file address.
qeh	4	Queue entry handle.
rq	varies	Request block. Size varies with the request.
sa	2	Segment address (high-order two bytes of a logical memory address) An sa may be either an sn or an sr. (See below.)
sg	2	Global Descriptor Table (GDT) selector.
sl	2	Local Descriptor Table (LDT) selector.
sn	2	Selector (high-order two bytes of a protected mode logical memory address).
sr	2	Paragraph number (high-order two bytes of a real mode logical memory address).
userNum	2	User number.

Suffixes

The suffix identifies the use of the variable. Suffixes are

Last	Largest allowable index of an array.
Max	Maximum length of an array or buffer (thus one greater than the largest allowable index).
Ret	Identifies a variable whose value is set by the called process or procedure rather than by the caller.

Examples of Variable Names

cbFileSpec	Count of bytes in a file specification.
ercRet	Error code to be returned to the caller.
pbFileSpec	Memory address of a string of bytes containing a file specification.
pDataRet	Memory address of an area to which data is to be returned to the caller.
ppDataRet	Memory address of a 4-byte memory area to which the memory address of a data item is returned to the caller.
pRq	Memory address of a request block.
sData	Size (in bytes) of a data area.
sDataMax	Maximum size (in bytes) of a data area.
psDataRet	Memory address of a 2-byte memory area to which the size of a data item is returned.
ssDataRet	Size of the memory area to which the size of a data item is returned.

1

Introduction to CTOS Operations

You can write programs that run on CTOS in several different languages. The discussion below is provided to introduce you to using CTOS operations in your programs. If you are new to the operating system, you should also Read "Using CTOS Operations" in the *CTOS Operating System Concepts Manual.* That chapter provides useful information on all the types of operations. For practical detailed programming examples showing how to use CTOS operations in your code, you should refer to the *CTOS/Open Programming Practices and Standards Manual* and the *CTOS Programming Guide.*

All operations are defined as "function subroutines." By definition, a function subroutine returns a value. Most operations return a one-word value called a status code; those that do not (for example, NPrint) are specifically identified in the text as not returning a code. (In these special cases, any value in the returned register is undefined and should be ignored.)

Status codes are commonly known as "ercs." When an operation returns a status code, it is labeled "ercType." If an ercType operation returns with no error, it returns a value of zero meaning "ercOK."

The operating system itself does not report any errors; it simply returns status codes. The calling operation is responsible for checking the returned status code and for doing error reporting or recovery.

Each operation is defined as follows:

OperationName(param-1, ..., param-n): type

where

OperationName

is the name of the operation.

param-n

is a parameter variable used in the operation.

type

is the type of the value returned (usually ercType or WORD).

The name of a parameter variable implies the type of parameter variable used in an operation. For example, variable names starting with the letter *p* refer to a pointer; variable names starting with the letter *b* refer to a byte; and most other variable names, including those starting with the letters *c*, *s*, and *i*, refer to a word value. For details, see the discussion on naming conventions in "About This Manual."

2
Operations by Function

Figure 2-1 shows all operations in this manual grouped by common function. Each functional group is presented alphabetically by function name. Within each group, the operations are categorized into subgroups, which are presented in a frequency-of-use order.

See Appendix J to determine whether an operation is supported on your version of the operating system.

BUS ADDRESS MANAGEMENT

DMA Buffer Management

DmaMapBuffer
DmaUnmapBuffer

Bus Addresses

MapBusAddress
RemapBusAddress
ReserveBusAddress
UnmapBusAddress

CACHER

CacheClose
CacheFlush
CacheGetEntry
CacheGetStatistics
CacheGetStatus
CacheInit
CacheReleaseEntry

CLUSTER MANAGEMENT

DisableCluster
FSrpUp
GetClstrGenerationNumber
GetClusterStatus
QueryWsNum

COMMUNICATIONS PROGRAMMING

AcquireByteStreamC
CheckForOperatorRestartC
CheckpointBsAsyncC
CheckPointBsC
DiscardInputBsC
DiscardOutputBsC
FillBufferAsyncC
FillBufferC
FlushBufferAsyncC
FlushBufferC
OpenByteStreamC
ReadByteStreamParameterC
ReadStatusC
ReleaseByteStreamC
SendBreakC
SetImageModeC
WriteByteStreamParameterC
WriteStatusC

DEMAND PAGING

QueryPagingStatistics

DEVICE DEPENDENT SAM

CloseVidFilter
DeleteByteStream
GetBsLfa
OpenVidFilter

Figure 2-1. Operations by Function (Page 1 of 19)

DEVICE DEPENDENT SAM (cont.)

PutBackByte
QueryVidBs
RenameByteStream
SetBsLfa
SetImageMode

DIRECT ACCESS METHOD

BASIC

CloseDaFile
DeleteDaRecord
OpenDaFile
ReadDaRecord
WriteDaRecord

ADVANCED

QueryDaRecordStatus
QueryDaLastRecord
ReadDaFragment
SetDaBufferMode
TruncateDaFile
WriteDaFragment

DISK MANAGEMENT

DismountVolume
Format
GetCtosDiskPartition

DISK MANAGEMENT (cont.)

MountVolume
QueryDeviceName
QueryDeviceNames
QueryDiskGeometry
SetDevParams
SetDiskGeometry

DYNAMIC LINK LIBRARIES

RUNTIME DYNAMIC LINKING

LibFree
LibGetHandle
LibGetInfo
LibGetProcInfo
LibLoad

INITIALIZING

ExitListSet
ExitListQuery

EXTENDED SYSTEM SERVICES

(See the *CTOS Programming Guide.)*

ASYNCHRONOUS SYSTEM SERVICE

AllocMemoryInit
AsyncRequest
AsyncRequestDirect
BuildAsyncRequest

Figure 2-1. Operations by Function (Page 2 of 19)

ASYNCHRONOUS SYSTEM SERVICE (cont.)

BuildAsyncRequestDirect
CheckContextStack
CreateContext
HeapAlloc
HeapFree
HeapInit
LogMsgIn
LogRespond
LogRequest
ResumeContext
SwapContextUser
TerminateAllOtherContexts
TerminateContext
TerminateContextUser

CD-ROM SERVICE

Volume Information

CdDirectoryList
CdGetDirEntry
CdGetVolumeInfo
CdSearchClose
CdSearchFirst
CdSearchNext
CdVerifyPath

Status

CdControl

Read File

CdClose
CdOpen
CdRead

Audio

CdAudioCtl

Other

CdAbsoluteRead
CdServiceControl

COMMAND ACCESS SERVICE

ObtainAccessInfo
ObtainUserAccessInfo

MOUSE SERVICES

Setup

PDInitialize
PDSetCharMapVirtualCoordinates
PDSetCursorType
PDSetMotionRectangle
PDSetMotionRectangleNSC
PDSetTracking
PDSetVirtualCoordinates

Figure 2-1. Operations by Function (Page 3 of 19)

Queries

GetIBusDevInfo
PdGetCursorPos
PDGetCursorPosNSC
PdQueryControls
PDQuerySystemControls
ReadInputEvent
ReadInputEventNSC

Cursor Shape

PdLoadCursor
PDLoadSystemCursor
PdReadCurrentCursor
PdReadIconFile

Cursor Movement

PDSetCursorDisplay
PDSetCursorPos
PDSetCursorPosNSC

Controls

PDSetControls
PDSetSystemControls

Transformations

PDTranslateVCToNSC
PDTranslateNSCToVC

System Programming in Multicontext Environment

PdAssignMouse

PERFORMANCE STATISTICS SERVICE

ClosePSSession
DeinstPSServer
GetPSCounters
OpenPSLogSession
OpenPSStatSession
PSCloseSession
PSDeinstServer
PSGetCounters
PSOpenLogSession
PSOpenStatSession
PSReadLog
PSResetCounters
ReadPSLog
ResetPSCounters

QUEUE MANAGER

AddQueue
AddQueueEntry
CleanQueue
DeinstallQueueManager
EstablishQueueServer
GetQMStatus
MarkKeyedQueueEntry
MarkNextQueueEntry
ReadKeyedQueueEntry
ReadNextQueueEntry
RemoveKeyedQueueEntry
RemoveMarkedQueueEntry
RemoveQueue
RescheduleMarkedQueueEntry
RewriteMarkedQueueEntry
TerminateQueueServer
UnmarkQueueEntry

Figure 2-1. Operations by Function (Page 4 of 19)

SEQUENTIAL ACCESS SERVICE

Basic

SeqAccessClose
SeqAccessControl
SeqAccessOpen
SeqAccessRead
SeqAccessStatus
SeqAccessWrite

Advanced

SeqAccessCheckpoint
SeqAccessDiscardBufferData
SeqAccessModeQuery
SeqAccessModeSet
SeqAccessRecoverBufferData

SPOOLER MANAGEMENT

ConfigureSpooler
SpoolerPassword

VOICE DATA SERVICES

Audio Service

AsGetVolume
AsSetVolume

Telephone Service

TsConnect
TsDataChangeParams
TsDataCheckpoint
TsDataCloseLine
TsDataOpenLine
TsDataRead
TsDataRetrieveParams
TsDataUnacceptCall
TsDataWrite
TsDeinstall
TsDial
TsDoFunction
TsGetStatus
TsHold
TsLoadCallProgressTones
TsOffHook
TsOnHook
TsQueryConfigParams
TsReadTouchTone
TsRing
TsSetConfigParams
TsVoiceConnect
TsVoicePlaybackFromFile
TsVoiceRecordToFile
TsVoiceStop

FILE MANAGEMENT

BASIC

CloseAllFiles
CloseFile

Figure 2-1. Operations by Function (Page 5 of 19)

BASIC (cont.)

OpenFile
Read
Write

BASIC UTILITY OPERATIONS

AtFileInit
AtFileNext
CreateFile
DeleteFile
RenameFile
WildCardClose
WildCardInit
WildCardNext

FILE ATTRIBUTES

ChangeFileLength
GetFileInfoByName
GetFileStatus
SetFileStatus

DEFAULT PATH

ClearPath
GetUcb
SetNode
SetPath
SetPrefix

DIRECTORIES

CreateDir
DeleteDir
GetDirInfo

DIRECTORIES (cont.)

GetDirStatus
ReadDirSector
SetDirStatus

LONG-LIVED FILES

CloseAllFilesLL
GetFhLongevity
OpenFileLL
SetFhLongevity

FILE HANDLE OPERATIONS

ChangeOpenMode
RemakeFh
ReopenFile

PARSING SPECIFICATIONS

ParseFileSpec
ParseSpecForDir
ParseSpecForFile
ParseSpecForNode
ParseSpecForPassword
ParseSpecForVol

BUILDING SPECIFICATIONS

BuildFileSpec
BuildSpecFromNode
BuildSpecFromVol
BuildSpecFromDir
BuildSpecFromFile
BuildSpecFromPassword
BuildFullSpecFromPartial

Figure 2-1. Operations by Function (Page 6 of 19)

PARSING AND BUILDING SPECIFICATIONS

GetCanonicalNodeAndVol
RemoveFfsBrackets

ASYNCHRONOUS FILE INPUT/OUTPUT

CheckReadAsync
CheckWriteAsync
ReadAsync
WriteAsync

VOLUME DATA STRUCTURES

GetVhb

INTER-CPU COMMUNICATION

GetBoardInfo
GetProcInfo
GetSlotFromName
GetSlotInfo
QueryBoardInfo
ReadKeySwitch
RemoteBoot
RequestRemote

INTERPROCESS COMMUNICATION

AllocExch
Check
DeallocExch

INTERPROCESS COMMUNICATION (cont.)

ForwardRequest
InstallNet
PSend
QueryDefaultResponseExch
Request
RequestDirect
Respond
Send
Wait
WaitLong

INTERRUPT HANDLERS

ControlInterrupt
DeviceInService
InitCommLine
MediateIntHandler
PSend
QueryTrapHandler
ReadCommLineStatus
ResetDeviceHandler
ResetTimerInt
ResetTrapHandler
ResetXBusMIsr
Set386TrapHandler
SetDefault386TrapHandler
SetDefaultTrapHandler
SetDeviceHandler
SetIntHandler
SetLdtRDs
SetLpIsr
SetTimerInt
SetTrapHandler
WriteCommLineStatus

Figure 2-1. Operations by Function (Page 7 of 19)

KEYBOARD AND I-BUS MANAGEMENT

BASIC

Beep
GetKbdId
GetKeyboardId
QueryKbdLeds
ReadKbd
ReadKbdDataDirect
ReadKbdDirect
ReadKbdInfo
SetKbdLed

ADVANCED

CheckpointSysIn
DisableActionFinish
KeyboardProfile
PostKbdTable
QueryKbdState
ReadActionCode
ReadActionKbd
ReadOsKbdTable
SetKbdUnencodedMode
SetKeyboardOptions
SetSysInMode
WriteKbdBuffer

I-BUS MANAGEMENT

ResetIBusHandler
SetIBusHandler
WriteIBusDevice
WriteIBusEvent

MAGNETIC CARD READER

PurgeMcr
ReadMcr

MEMORY MANAGEMENT

SHORT-LIVED MEMORY

AllocAllMemorySL
AllocAreaSL
AllocMemoryFramesSL
AllocMemorySL
DeallocMemorySL
ExpandAreaSL
ShrinkAreaSL

LONG-LIVED MEMORY

AllocMemoryLL
DeallocMemoryLL
ExpandAreaLL
ResetMemoryLL
ShrinkAreaLL

SHORT-LIVED AND LONG-LIVED MEMORY

DefineInterLevelStack
DefineLocalPageMap
QueryBigMemAvail
QueryMemAvail
ShrinkPartition

Figure 2-1. Operations by Function (Page 8 of 19)

I/O MANAGEMENT

QueryIOOwner
ResizeIOMap
SetIOOwner

SELECTOR MANAGEMENT

CreateAlias
DeallocSg
DeallocAliasForServer
FValidPbCb
GetSegmentLength
LaFromP
LaFromSn
RemakeAliasForServer
ReuseAlias
ReuseAliasLarge
SetSegmentAccess
SgFromSa
SnFromSr
SrFromSn

GLOBAL LINEAR ADDRESS SPACE MANAGEMENT

AllocateSegment
AllocHugeMemory
DeallocateSegment
DeallocHugeMemory
ReallocHugeMemory
ResizeSegment

NATIVE LANGUAGE SUPPORT

NLS UTILITY

GetNlsDateName
GetNlsDateTimeTemplate
GetNlsKeycapText
GetNlsTable
GetPNlsTable
NlsCase
NlsClass
NlsCollate
NlsFormatDateTime
NlsGetYesNoStrings
NlsGetYesNoStringSize
NlsNumberandCurrency
NlsParseTime
NlsSpecialCharacters
NlsStandardize
NlsStdFormatDateTime
NlsULCmpB
NlsVerifySignatures
NlsYesNoOrBlank
NlsYesOrNo
OpenNlsFile
ULCmpB

Figure 2-1. Operations by Function (Page 9 of 19)

ENLS UTILITY

Character Processing

EnlsCase
EnlsClass
EnlsGetChar
EnlsGetCharWidth
EnlsGetPrevChar
EnlsMapCharToStdValue
EnlsMapStdValueToChar

String Processing

EnlsAppendChar
EnlsDeleteChar
EnlsFieldEdit
EnlsFieldEditByChar
EnlsFindC
EnlsFindRC
EnlsInsertChar

Line/Form Drawing

EnlsCbToCCols
EnlsDrawBox
EnlsDrawFormChars
EnlsDrawLine
EnlsQueryBoxSize

MESSAGE FILE

Standard Operations

CloseMsgFile
ExpandLocalMsg
GetMsg

Standard Operations (cont.)

GetMsgUnexpanded
GetMsgUnexpandedLength
InitMsgFile
PrintMsg

Alternate Operations

CloseAltMsgFile
GetAltMsg
GetAltMsgUnexpanded
GetAltMsgUnexpandedLength
InitAltMsgFile
PrintAltMsg

System Service Operations

CloseServerMsgFile
GetServerMsg
OpenServerMsgFile

PARAMETER MANAGEMENT

CONSTRUCTING PARAMETERS

RgParamInit
RgParamSetEltNext
RgParamSetListStart
RgParamSetSimple

QUERYING PARAMETERS

CParams
CSubParams
RgParam

Figure 2-1. Operations by Function (Page 10 of 19)

DEALLOCATING VLPB

DeallocVlpb

PROCESS MANAGEMENT

ChangePriority
ChangeProcessPriority
CreateProcess
NewProcess
QueryProcessNumber
RescheduleProcess
SetDeltaPriority
SetDispMsw287
SuspendUser
UnsuspendUser
SuspendProcess
UnsuspendProcess

PARTITIONS AND PARTITION MANAGEMENT

BASIC OPERATIONS

GetPartitionHandle
GetPartitionStatus
GetUserNumber
SetPartitionName

SWAPPING

GetPartitionSwapMode
LockInContext

SWAPPING (cont.)

SetPartitionSwapMode
SetSwapDisable
SwapInContext
UnlockInContext

PARTITION CREATION UNDER PROGRAM CONTROL

AssignKbd
AssignVidOwner
CreateBigPartition
CreatePartition
CreateUser
ReservePartitionMemory

PARTITION LOADING UNDER PROGRAM CONTROL

LoadPrimaryTask
LoadInteractiveTask
RemovePartition
SetPartitionLock
TerminatePartitionTasks
VacatePartition

LOADING ADDITIONAL TASKS

DeallocRunFile
LoadRunFile
LoadTask

Figure 2-1. Operations by Function (Page 11 of 19)

PROGRAM MANAGEMENT

ERROR HANDLING

CheckErc
Crash
ErrorExit
ErrorExitString
FatalError
SetMsgRet

NORMAL PROGRAM Exit

Chain
Exit
ExitAndRemove
QueryExitRunFile
SetExitRunFile

RECORD SEQUENTIAL ACCESS METHOD

BASIC

CloseRsFile
OpenRsFile
ReadRsRecord
WriteRsRecord

ADVANCED

CheckpointRsFile
GetRsLfa
ReleaseRsFile
ScanToGoodRsRecord
SetRsLfa

SCSI DEVICE MANAGEMENT

SCSI PATHS

GetScsiInfo
ScsiManagerNameQuery
ScsiOpenPath
ScsiQueryInfo
ScsiQueryPathParameters
ScsiSetPathParameters
ScsiClosePath

BASIC I/O

ScsiCdbDataIn
ScsiCdbDataOut
ScsiRequestSense

ADVANCED

ScsiCdbDataInAsync
ScsiCdbDataOutAsync
ScsiWaitCdbAsync
ScsiReset

TARGET MODE

ScsiTargetCdbCheck
ScsiTargetCdbWait
ScsiTargetDataReceive
ScsiTargetDataReceiveAsync
ScsiTargetDataTransmit
ScsiTargetDataTransmitAsync
ScsiTargetOperationsAbort
ScsiWaitTargetDataAsync

Figure 2-1. Operations by Function (Page 12 of 19)

SEMAPHORES

SYSTEM SEMAPHORES

SemClose
SemOpen

MUTUAL EXCLUSION

SemClear
SemLock

CRITICAL SECTION

SemClearCritical
SemLockCritical

SIGNALING AND SYNCHRONIZATION

SemClear
SemMuxWait
SemSet
SemWait

SEQUENTIAL ACCESS METHOD

BASIC

CloseByteStream
GetBsMember
OpenByteStream
OutputToVid0
ReadBsRecord

BASIC (cont.)

ReadByte
WriteBsRecord
WriteByte

ADVANCED

CheckpointBs
QueryVidBs
ReadBytes
ReleaseByteStream

SERIAL PORT MANAGEMENT

ChangeCommLineBaudRate
GetCommLineDmaStatus
InitCommLine
LockIn
LockOut
ReadCommLineStatus
ReceiveCommLineDma
ResetCommLine
TransmitCommLineDma
WriteCommLineStatus

SRP TERMINAL MANAGEMENT

CloseTerminal
DrainTerminalOutput
OpenTerminal
ReadTerminal
SetTerminal
WhereTerminalBuffer

Figure 2-1. Operations by Function (Page 13 of 19)

STRUCTURED FILE ACCESS METHODS

GetStamFileHeader

SYSTEM DEFINITIONS

CLUSTER MANAGEMENT

GetClusterStatus

DISK MANAGEMENT

QueryDiskGeometry
SetDiskGeometry

FILE MANAGEMENT

GetVhb

OPERATING SYSTEM

CurrentOsVersion
EnterBootRom
FilterDebugInterrupts
FMergedOs
FProcessorSupportsProtectedMode
FProtectedMode
FRmos
FRmosUser
FVSeries
GetCoprocessorStatus
GetFRmosUser
GetLocalDaiNumber

OPERATING SYSTEM (cont.)

GetPartitionStatus
GetPAscb
GetPStructure
OsVersion
QueryCoprocessor
QueryLdtR
QueryNodeName
SerialNumberOldOsQuery
SerialNumberQuery
SetPStructure

USER NAME MANAGEMENT

GetUcb
GetUserStatus
GetWsUserName
QueryUserLocation
SetWsUserName

VIDEO

QueryVideo
QueryVidHdw

SYSTEM-COMMON SERVICES MANAGEMENT

SystemCommonInstall
SystemCommonQuery

Figure 2-1. Operations by Function (Page 14 of 19)

SYSTEM SERVICES MANAGEMENT

BASIC REQUESTS

ConvertToSys
QueryNodeName
QueryRequestInfo
ServeRq
SetPartitionName

SYSTEM REQUESTS

AbortRequests
ChangeUserNumberRequests
SwappingRequests
TerminationRequests

TIMER MANAGEMENT

DELAY

Delay
Doze
ShortDelay

REALTIME CLOCK

CloseRtClock
OpenRtClock

PROGRAMMABLE INTERVAL TIMER

ResetTimerInt
SetTimerInt

UTILITY OPERATIONS

ACCESSING RESOURCES IN DISK FILES

RsrcCopyFromFile
RsrcDelete
RsrcEndSetAccess
RsrcGetAllSetTypeInfo
RsrcGetCountAllSetType
RsrcGetCountSetType
RsrcGetDesc
RsrcGetSetTypeInfo
RsrcInitSetAccess
RsrcSessionEnd
RsrcSessionInit

ACCESSING RESOURCES IN MEMORY

GetModuleResourceInfo

COMPARING STRINGS

FComparePointer
FsCanon
NlsGetYesNoStrings
NlsGetYesNoStringSize
NlsULCmpB
NlsYesNoOrBlank
NlsYesOrNo
StringsEqual
ULCmpB
WildCardMatch

Figure 2-1. Operations by Function (Page 15 of 19)

CUSTOMIZING THE USER INTERFACE

EnIsFieldEdit
EnIsFieldEditByChar
FieldEdit
FormEdit
MenuEdit
QueryZoomBoxPosition
QueryZoomBoxSize
UnzoomBox
ZoomBox

DIRECTING DATA TO A BYTE STREAM

NPrint
OutputBytesWithWrap
OutputQuad
OutputWord
PrintFileClose
PrintFileOpen
PrintFileStatus
PutByte
PutChar
PutPointer
PutQuad
PutWord
SbPrint
ZPrint

HANDLING COMMANDS

CloseSysCmds
GetSysCmdInfo
InitSysCmds

MANAGING NAMES

NameAllocClass
NameQuery
NameRegister
NameRemove

MANIPULATING ERROR MESSAGES

CloseErcFile
GetErc
GetErcLength
GetFileErc
GetStandardErcMsg
InitErcFile
PrintErc

OBTAINING THE SYSTEM DATE AND TIME

CompactDateTime
ExpandDateTime
FormatDateTime
FormatTime
FormatTimeDt
FormatTimeTm
GetDateTime
NIsFormatDateTime
NIsParseTime
NIsStdFormatDateTime
ParseTime
SetDateTime
SetDateTimeMode

Figure 2-1. Operations by Function (Page 16 of 19)

OBTAINING VERSIONS OF SYSTEM SERVICES

AccessVersion
CdVersion
CfaFFVersion
CfaServerVersion
CfaWaVersion
DcxVersion
McrVersion
MouseVersion
QueueMgrVersion
RkvsVersion
ScreenPrintVersion
SeqAccessVersion
SpoolerVersion
StatisticsVersion
TsVersion
XbifVersion
XC002Version

PARSING CONFIGURATION FILES

Parsing Nonstandard Configuration Files

ConfigCloseFile
ConfigGetNextToken
ConfigGetPosition
ConfigGetRestOfLine
ConfigOpenFile
ConfigSetPosition

Parsing Standard Configuration Files

LookUpField
LookUpNumber
LookUpReset

Parsing Standard Configuration Files (cont.)

LookUpString
LookUpValue
OpenUserFile
ReadToNextField
SetField
SetFieldNumber

Parsing User Configuration Files

GetUserFileEntry
SetUserFileEntry

USING WORKSTATION HARDWARE IDS

GetClusterId
OldReadHardId
OldWriteHardId
ReadHardId
WriteHardId

PERFORMING MISCELLANEOUS TASKS

CopyFile
CreateExecScreen
GenResString
QueryMail
SignOnLog
TextEdit
WriteLog

Figure 2-1. Operations by Function (Page 17 of 19)

VIDEO

VAM OPERATIONS

FillFrame
FillFrameRectangle
FrameBackSpace
MoveFrameRectangle
PosFrameCursor
PutCharsAndAttrs
PutFrameAttrs
PutFrameChars
PutFrameCharsAndAttrs
QueryBounds
QueryCharsAndAttrs
QueryCursor
QueryFrameBounds
QueryFrameChar
QueryFrameCharsAndAttrs
QueryFrameCursor
ScrollFrame

VIDEO REQUESTS

QueryFrameAttrs
QueryFrameString

VDM OPERATIONS

InitCharMap
InitVidFrame
LoadFontRam
QueryVideo

VDM OPERATIONS

QueryVidHdw
ResetVideo
ResetVideoGraphics
ResetFrame
SetScreenVidAttr
SetVideoTimeout

COLOR PROGRAMMING

LoadColorStyleRam
LoadBackgroundPalette
ProgramColorMapper
SetAlphaColorDefault
SetStyleRam
SetStyleRamEntry

DIRECT ACCESS TO VIDEO DATA STRUCTURES

LockVideo
LockVideoForModify
UnlockVideo
UnlockVideoForModify

VIRTUAL CODE MANAGEMENT

BASIC

InitLargeOverlays
InitOverlays

Figure 2-1. Operations by Function (Page 18 of 19)

ADVANCED

DeallocateRods
EnableSwapperOptions
GetCParaOvlyZone
GetOvlyStats
MakePermanent
MakePermanentP
MakeRecentlyUsed
MapCslOvly
MaplOvlyCs
MapPStubPProc
MoveOverlays
ReinitLargeOverlays
ReinitOverlays
ReleasePermanence
ReinitStubs
UpdateOverlayLru

X-BUS MANAGEMENT

GetModuleAddress
GetModuleId
LockXbis
MapPhysicalAddress
MapXBusWindow
MapXBusWindowLarge
Mode3DmaReload
QueryModulePosition
ResetXBusMIsr
SetModuleId
SetXBusMIsr
SwapXBusEar
UnlockXbis
UnmapPhysicalAddress

Figure 2-1. Operations by Function (Page 19 of 19)

3
Operations

This chapter lists alphabetically all the CTOS operations available to programs. To determine if a particular operation is supported on your operating system, refer to Appendix J, "Operation Data."

This page intentionally left blank

Abort requests have no procedural interface.

NOTE: This page presents the standard format for all abort requests.

Description

Abort requests function similarly to termination requests in that they notify all system services that a client program has terminated. Upon notification, system services can release resources, such as open files and locked ISAM records, that have been allocated to the terminating program.

The operating system issues abort requests when

- the server detects that it cannot communicate with a cluster workstation
- a partition is vacated with the VacatePartition operation or through lack of an exit run file

(See Appendix D for a list of the operating system abort requests and request codes that are issued.)

If optional system services are installed that define other abort requests, the operating system issues these requests along with those listed in Appendix D.

In addition to abort requests, the operating system issues termination requests on a workstation when the program terminates or is terminated. (See the TerminationRequests operation page in this chapter for more information.)

Procedural Interface

Abort requests have no procedural interface.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	*
12	ercAbort	2	8200

*Request code of abort request

AccessVersion (pbNode, cbNode, pVerStruct, sVerStruct, pcbRet): ercType

Description

AccessVersion returns a structure that contains the version of the installed Command Access Service, as well as relevant status information.

Procedural Interface

AccessVersion (pbNode, cbNode, pVerStruct, sVerStruct, pcbRet): ercType

where

pbNode
cbNode

describe a string of the form {Node}, where Node is the name of the node on which the Command Access Service is installed.

pVerStruct
sVerStruct

describe a memory area to which the structure containing version information is returned. The format of the structure is as follows:

Offset	Field	Size (Bytes)
0	version	2
2	qnLogSectors	4
6	fChecking	1
7	fLogging	1

pcbRet

describes a word where the actual length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	821
12	reserved	6	
18	pbNode	4	
22	cbNode	2	
24	pVerStruct	4	
28	sVerStruct	2	
30	pcbRet	4	
34	scbRet	2	2

AcquireByteStreamC (pBswa, pbDevSpec, cbDevSpec, mode, pConfig, pbTQ, cbTQ, pbRQ, cbRQ, cbRxMax,bSpecial): ercType

Description

AcquireByteStreamC is a substitute for OpenByteStreamC. It does not require a configuration file on disk and offers more flexibility.

(See "Communications Programming" in the *CTOS Operating System Concepts Manual* for more information.)

Procedural Interface

AcquireByteStreamC (pBswa, pbDevSpec, cbDevSpec, mode, pConfig, pbTQ, cbTQ, pbRQ, cbRQ, cbRxMax,bSpecial): ercType

where

pBswa

is the memory address of the Byte Stream Work Area (BSWA). Note that for Standard Software versions 12.1 or later, 160 bytes are required for this BSWA. For versions 12.0 or earlier, this BSWA requires 130 bytes.

pbDevSpec

is the memory address of the "[Comm]" or "[Ptr]" device specification. Anything that follows an ampersand (&) is ignored.

cbDevSpec

is the size of the device specification in characters (word).

mode

is a code as follows:

mr or mt	mode read
mw or ma	mode write
mm	mode modify (read and write)

pConfig

is the memory address of the configuration descriptor. It may be overlayed with the transmit and/or receive queue areas. (For details, see "Communications Configuration Descriptor" in Chapter 4, "System Structures.")

pbTQ

is the address in memory of the actual transmit queue area. *pbTQ* may be 0 for read-only modes.

cbTQ

is the size of the transmit queue (word). The first 12 bytes are used for control information; the rest is the maximum *cbRet* that FlushBufferC ever returns. The transmitting process is awakened at least every (*cbTQ*-12)/2 bytes. *cbTQ* may be 0 for read-only modes.

pbRQ

is the memory address of the actual receive area. If this area overflows, status code 2346 ("Receive queue overrun") is returned, and data is lost. *pbRQ* may be 0 for write-only modes.

cbRQ

is the size of the receive queue (word). The first 12 bytes are used for control information. The receiving process is awakened at least every (*cbRQ*-12)/2 bytes. *cbRQ* may be 0 for write-only modes.

cbRxMax

is the maximum cbRet that FillBufferC will ever be allowed to return (word). *cbRQ*/2 is an appropriate choice for the value of *cbRxMax*. If *cbRxMax* is too large, status code 2346 ("Receive queue overrun") may be returned and data may be lost. Since FillBufferC does not copy the data, *cbRet* bytes of the receive queue are unavailable (to receive new characters) until the next FillBufferC.

bSpecial

controls the operation of Acquire/Release calls (byte). *bSpecial* is for internal use only and must be 0.

Request Block

AcquireByteStreamC is an object module procedure.

This page intentionally left blank

AddQueue(pbQueueName, cbQueueName, pbFileSpec, cbFileSpec, wEntrysize, wQueueType, pQhRet): ercType

Description

AddQueue dynamically activates a queue, enabling the Queue Manager to add a queue to its table and to create and/or open the corresponding queue entry file. The parameters passed are the same as would be seen when reading the Queue Index File, and a queue handle is returned for future reference. This operation allows a new queue to be made operational dynamically without configuration files or system reboot.

Procedural Interface

AddQueue (pbQueueName, cbQueueName, pbFileSpec, cbFileSpec, wEntrysize, wQueueType, pQhRet): ercType

where

pbQueueName
cbQueueName

are a user-defined queue name that is unique to the installation. The name can be any name less than 50 characters.

pbFileSpec
cbFileSpec

describe the file specification of the queue entry file where queue entries submitted by programs requesting to add entries are stored.

wEntrysize

is the size of an entry for the queue entry file. The size is the number of 512-byte sectors per entry.

wQueueType

is the type of the queue (an integer less than or equal to 255).

pQhRet

specifies the one-word area where the queue handle is written.

Request Block

sQhRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32865
12	wEntrySize	2	
14	wQueueType	2	
16	pbQueueName	4	
20	cbQueueName	2	
22	pbFileSpec	4	
26	cbFileSpec	2	
28	pQhRet	4	
32	sQhRet	2	2

AddQueueEntry (pbQueueName, cbQueueName, fQueueIfNoServer, priority, queueType, pEntry, sEntry, pDateTime, repeatTime): ercType

Description

AddQueueEntry is used by the caller to add an entry to the specified queue for processing by the appropriate queue server.

Procedural Interface

AddQueueEntry (pbQueueName, cbQueueName, fQueueIfNoServer, priority, queueType, pEntry, sEntry, pDateTime, repeatTime): ercType

where

pbQueueName
cbQueueName

describe a queue name.

fQueueIfNoServer

is a flag. TRUE means the Queue Manager adds the entry to the specified queue whether or not programs have been established as queue servers for the queue. FALSE means the Queue Manager returns status code 908 ("Queue not served") if no program has been established as the queue server for the queue.

priority

is the priority (0-9, with 0 the highest) at which the entry is placed in the queue.

queueType

is the type of the queue (an integer less than or equal to 255), which is used in a consistency check. The Queue Manager checks the type of the queue described by the queue name against this parameter.

pEntry
sEntry

describe the buffer that contains the type-specific portion of the queue entry.

pDateTime

is the memory address of the 32-bit date/time in Convergent format. (See "Timer Management" in the *CTOS Operating System Concepts Manual* for more information.) *DateTime* specifies the earliest time the queue entry can be processed by a program serving the queue. If *DateTime* is equal to 0, the entry can be processed immediately.

repeatTime

specifies the repeating time interval in minutes (up to 65,335 minutes) at which the queue entry is to be processed by a queue server. A 0 means no repetition occurs. (For example, to repeat the entry once a day, specify 1,440 minutes; to repeat the entry once each week, specify 10,080 minutes.)

Request Block

sDateTime is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	137
12	fQueueIfNoServer	1	
13	priority	1	
14	queueType	2	
16	repeatTime	2	
18	pbQueueName	4	
22	cbQueueName	2	
24	pEntry	4	
28	sEntry	2	
30	pDateTime	4	
34	sDateTime	2	4

This page intentionally left blank

AllocAllMemorySL (pcParagraphRet, ppSegmentRet): ercType

Description

AllocAllMemorySL creates a short-lived segment and allocates all free memory available to the program.

In protected mode, if the size of allocated memory is greater than 64K bytes, the region beyond 64K bytes is not addressable using the returned memory address.

For this reason, programs designed to execute in both real and protected modes should use the following short-lived memory allocation operations:

- AllocMemorySL
- AllocAreaSL
- ExpandAreaSL

Procedural Interface

AllocAllMemorySL (pcParagraphRet, ppSegmentRet): ercType

where

pcParagraphRet

is the memory address of a word to which the count of bytes available (divided by 16) is returned.

ppSegmentRet

is the memory address into which the 4-byte memory address of the allocated segment is returned. The offset, which is always 0, is returned in the low-order two bytes. The segment address is returned in the high-order two bytes.

Request Block

scParagraphMax is always 2 and *spSegmentMax* is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	46
12	reserved	6	
18	pcParagraphRet	4	
22	scParagraphMax	2	2
24	ppSegmentRet	4	
28	spSegmentMax	4	4

AllocAreaSL (cBytes, ppSegmentRet): ercType

Description

AllocAreaSL creates a short-lived segment and allocates memory of the specified size at the end of the segment.

This operation differs from AllocMemorySL in that the offset portion of the allocated segment's address is not necessarily 0. (See AllocMemorySL.)

In protected mode, this operation also allocates a new data segment descriptor of the expand-down type.

Programs designed to execute in both real and protected modes should use AllocAreaSL if the program will allocate additional memory in the short-lived segment. A program should use ExpandAreaSL to allocate additional memory.

Procedural Interface

AllocAreaSL (cBytes, ppSegmentRet): ercType

where

cBytes

is the count of bytes to be allocated.

ppSegmentRet

is the memory address into which the address of the allocated memory within the segment is returned. The offset, (not necessarily 0), is returned in the low order 2 bytes. The segment address is returned in the high order 2 bytes.

Request Block

spSegmentMax is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	278
12	cBytes	2	
14	reserved	4	
18	ppSegmentRet	4	
22	spSegmentMax	2	4

AllocateSegment (qBytes, fExpandDown, p48Ret): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

AllocateSegment creates a short-lived segment of the specified size in the linear address space. The segment must be sized in increments of 4K bytes.

Procedural Interface

AllocateSegment (qBytes, fExpandDown, p48Ret): ercType

where

qBytes

is the size (dword) of the segment rounded up to the next 4K byte boundary.

fExpandDown

is a flag (word). TRUE means the segment is an expand down segment. Otherwise, the segment is expand up.

p48Ret

is the memory address of the location to which the 48-bit segmented address of the allocated memory is returned. The low-order 32 bits are the offset of the allocated memory in the segment. The high-order 16 bits contain the segment selector.

Request Block

s48Ret is always 6.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	433
12	qBytes	4	
16	fExpandDown	2	
18	p48Ret	4	
22	s48Ret	2	6

AllocCommDmaBuffer (cbBuffer, ppBufferRet, pbaBufferRet): ercType

Description

AllocCommDmaBuffer is called to allocate a memory buffer suitable for DMA communications.

Procedural Interface

AllocCommDmaBuffer (cbBuffer, ppBufferRet, pbaBufferRet): ercType

where

cbBuffer

is the desired segment size.

ppBufferRet

is the memory address into which the 4-byte memory address of the allocated segment is returned. The offset,which is always 0, is returned in the low order two bytes. The segment address is returned in the high order two bytes.

pbaBufferRet

is an optional parameter. If a non-zero pointer is passed, the buffer address is returned in a form that is suitable for programming DMA hardware. On a shared resource processor, obtaining this address causes hardware resources to be allocated, the returned address is a bus address. On hardware other than a shared resource processor, the returned address is a physical address.

Request Block

AllocCommDmaBuffer is an object-module procedure.

This page intentionally left blank.

AllocExch (pExchRet): ercType

Description

AllocExch allocates an exchange.

Procedural Interface

AllocExch (pExchRet): ercType

where

pExchRet

is the memory address of a word to which the identification number of the allocated exchange is returned. On a shared resource processor, this exchange number is unique across all processor boards.

Request Block

sExchMax is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	40
12	reserved	6	
18	pExchRet	4	
22	sExchMax	2	2

This page intentionally left blank

AllocHugeMemory (cb, cbMax, fSharable, pSnRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

AllocHugeMemory allocates short-lived memory of the specified size in the linear address space and returns the first of a sequence of expand up selectors to address the memory. Each successive selector has a numeric value 8 greater than the previous selector.

Each allocated selector except the last addresses 64K bytes. The last selector addresses 64K bytes or less.

The memory allocated by this operation may be subsequently expanded up to the specified maximum size using the ReallocHugeMemory operation. (See ReallocHugeMemory.)

Procedural Interface

AllocHugeMemory (qb, qbMax, fSharable, pSnRet): ercType

where

qb

is the desired memory size (dword).

qbMax

is the maximum size (dword) to which the allocated memory may be expanded.

fSharable

is a flag (word). TRUE means the allocated memory can be addressed by another user number using the returned selectors.

pSnRet

is the memory address into which the first selector that addresses the allocated memory is returned.

Request Block

sSnRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	10
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	450
12	fSharable	2	
14	cb	4	
18	cbMax	4	
22	pSnRet	4	
26	sSnRet	2	2

AllocMemoryFramesSL (cFrames, ppSegmentRet): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

Description

AllocMemoryFramesSL creates a short-lived segment and allocates cFrames*4096 bytes of short-lived memory at the beginning of the segment.

The beginning of the segment is aligned on a 4K-byte boundary in physical memory.

AllocMemoryFramesSL supports system software that requires a 4K-byte alignment for data structures. For example, page maps used by the 80386 microprocessor require a 4K-byte alignment.

Procedural Interface

AllocMemoryFramesSL (cFrames, ppSegmentRet): ercType

where

cFrames

is the number of frames in the segment. A frame is 4096 bytes.

ppSegmentRet

is the memory address into which the memory address of the allocated segment is returned.

Request Block

spSegmentRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCp	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	325
12	cFrames	2	
14	reserved	4	
18	ppSegmentRet	4	
22	spSegmentRet	2	4

AllocMemoryInit (cBytes, ppMemoryRet, fInit): ercType

Description

AllocMemoryInit allocates memory out of the DS space below the last CODE segment. The common-code module of the asynchronous system service model uses this procedure to allocate memory addressable by short pointers.

Procedural Interface

AllocMemoryInit (cBytes, ppMemoryRet, fInit): ercType

where

cBytes

is is the count of bytes to be allocated.

ppMemoryRet

is the memory address to which the pointer to the DS space allocated is returned.

fInit

is a flag that is normally FALSE. *fInit* is TRUE only if the call is being made from the asynchronous system service initialization module InitAlloc (or any user-created module containing initialization code to be deallocated after use) and the data allocated will be initialized or used before all such modules have finished executing. (These modules are referred to as COED modules. For additional information, see the *CTOS Programming Guide*.) Setting this flag prevents code yet to be executed from being overwritten. AllocMemoryInit returns status code 26 ("Stray interrupt") if *fInit* is TRUE and the next space to allocate is COED space. To avoid this problem, your program should allocate memory after all COED modules have finished executing.

Request Block

AllocMemoryInit is an object module procedure in Async.lib.

AllocMemoryLL (cBytes, ppSegmentRet): ercType

Description

AllocMemoryLL creates a long-lived segment and allocates memory of the specified size at the beginning of the segment.

In protected mode, this operation also allocates a new data segment descriptor of the expand-up type.

Programs designed to execute in both real and protected modes should use ExpandAreaLL to allocate additional memory within the segment created by AllocMemoryLL.

Procedural Interface

AllocMemoryLL (cBytes, ppSegmentRet): ercType

where

cBytes

is the desired segment size.

ppSegmentRet

is the memory address into which the 4-byte memory address of the allocated segment is returned. The offset, which is always 0, is returned in the low-order two bytes. The segment address is returned in the high-order two bytes.

Request Block

spSegmentMax is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp		
8	ercRet	2	
10	rqCode	2	44
12	cBytes	2	
14	reserved	4	
18	ppSegmentRet	4	
22	spSegmentMax	2	4

AllocMemorySL (cBytes, ppSegmentRet): ercType

Description

AllocMemorySL creates a short-lived segment and allocates memory of the specified size at the beginning of the segment. AllocMemorySL differs from AllocAreaSL in that the offset portion of the allocated segment's address is always 0, enabling the segment to be addressed using only the 16-bit segment address. (See AllocAreaSL.)

In protected mode, this operation also allocates a new data segment descriptor of the expand-up type.

Programs designed to execute in both real and protected modes should use this operation only if additional memory will not be allocated in the segment. A program should use AllocAreaSL if additional memory will be allocated.

Procedural Interface

AllocMemorySL (cBytes, ppSegmentRet): ercType

where

cBytes

is the desired segment size.

ppSegmentRet

is the memory address into which the 4-byte memory address of the allocated segment is returned. The offset, which is always 0, is returned in the low-order two bytes. The segment address is returned in the high-order two bytes.

Request Block

spSegmentMax is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	42
12	cBytes	2	
14	reserved	4	
18	ppSegmentRet	4	
22	spSegmentMax	2	4

AllocUserNumbers (cUserNum, pUserNumBlockRet): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

Description

AllocUserNumbers allocates a specified quantity of user numbers for use by an internal system service. The user numbers are returned to the caller in the format of a user number block at the specified address. User numbers allocated by this operation are deallocated with DeallocUserNumbers.

Procedural Interface

AllocUserNumbers (cUserNum, pUserNumBlockRet): ercType

where

cUserNum

is the number (word) of user numbers to allocate.

pUserNumBlockRet

is the memory address into which the user number block structure is returned. The structure has the following format:

Offset	Field	Size (bytes)	Description
0	userNumFirst	2	First user number in block
2	userNumLast	2	Last user number in block
4	blockID	2	ID of this block of user numbers

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	308
12	cUserNum	2	
14	pUserNumBlockRet	4	
18	sUserNumBlockRet	2	

AsGetVolume (iDevice, bVolumeType, pVolumeRet): ercType

Caution: *AsGetVolume can only be used on SuperGen Series 5000 workstations.*

Description

AsGetVolume returns the volume level at which the Video/Voice Cartridge plays back messages or emits system indicators (for example, an error beep).

Although the volume level ranges from 0 to 1024, levels above 256 should be selected with caution because signal distortion may result.

An application can use AsGetVolume to determine if the volume level is too high or too low. The application can then change the volume setting to the desired level by calling AsSetVolume.

Status code 11203 ("Invalid volume level") is returned if the volume level is greater than 1024.

This operation is performed by the Audio Service. (See "Voice/Data Services" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

AsGetVolume (iDevice, bVolumeType, pVolumeRet): ercType

where

iDevice

is the Video/Voice Cartridge number. This value must currently be 1.

bVolumeType

is always 0.

pVolumeRet

is the memory address of a word to which the current volume setting is returned.

Request Block

iDevice is always 1, *bVolumeType* is always 0, and *sVolumeRet* is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33171
12	iDevice	2	1
14	bVolumeType	1	0
15	reserved	1	
16	pVolumeRet	4	
20	sVolumeRet	2	2

This page intentionally left blank

AsSetVolume (iDevice, bVolumeType, wVolume): ercType

Caution: *AsSetVolume can only be used on SuperGen Series 5000 workstations.*

Description

AsSetVolume sets the volume level at which the Video/Voice Cartridge plays back messages or emits system indicators (for example, an error beep).

Although the volume level ranges from 0 to 1024, levels above 256 should be selected with caution because signal distortion may result.

Before an application plays back messages, it can call AsGetVolume to determine if the volume level is too high or too low. If the volume needs to be changed, the application can then call AsSetVolume.

Status code 11203 ("Invalid request parameter") is returned if the value specified in the *bVolumeType* parameter is invalid. This status code is also returned if an out-of-range value is specified in the *wVolume* parameter when *bVolumeType* is 1.

This operation is performed by the Audio Service. (See "Voice/Data Services" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

AsSetVolume (iDevice, bVolumeType, wVolume): ercType

where

iDevice

is the Video/Voice Cartridge number. Currently, this value must be 1.

bVolumeType

is the type of volume operation to be performed. Each value indicates the following:

Value	Description
1	Set the volume to the level specified by the *wVolume* parameter.
2	Increment the volume by one step. (The step size is 16.)
3	Decrement the volume by one step.

wVolume

is the volume level at which the Video/Voice Cartridge is to be set. This value need only be specified when the *bVolumeType* parameter value is 1. If *wVolume* is less than 0 or greater than 1024, the volume is not changed, and status code 11203 ("Invalid request parameter") is returned.

Request Block

iDevice is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33174
12	iDevice	2	1
14	bVolumeType	1	
15	reserved	1	
16	wVolume	2	

This page intentionally left blank

AssignKbd (userNum): ercType

Description

AssignKbd assigns the keyboard to an application partition.

This operation should not be used by programs that run under a partition-managing program, such as the Context Manager.

Procedural Interface

AssignKbd (userNum): ercType

where

userNum

is the user number returned from a CreatePartition or a GetPartitionHandle operation.

Request Block

AssignKbd is a system-common procedure.

This page intentionally left blank

AssignVidOwner (userNum):ercType

Caution: *For compatibility on all workstations, it is recommended that you use the Video Access Method (VAM) rather than using this operation.*

Description

AssignVidOwner assigns the screen to an application partition.

This operation should not be used by programs that run under a partition-managing program, such as the Context Manager.

Procedural Interface

AssignVidOwner (userNum):ercType

where

userNum

is the user number returned from a CreatePartition or a GetPartitionHandle operation.

Request Block

AssignVidOwner is a system–common procedure.

This page intentionally left blank

AsyncRequest (pRq): ercType

Description

AsyncRequest is called by an asynchronous system service to send a request block to the operating system for routing to the appropriate service exchange. To send the request, AsyncRequest uses the Kernel primitive request. AsyncRequest then returns to the top of the asynchronous system service wait loop. When the response arrives, the context is resumed at the next program statement.

Procedural Interface

AsyncRequest (pRq): ercType

where

pRq

is the memory address of the request block.

Request Block

AsyncRequest is an object module procedure in Async.lib.

This page intentionally left blank

AsyncRequestDirect (exch, pRq): ercType

Description

AsyncRequestDirect is called by an asynchronous system service to send the request block to the operating system for routing to the appropriate service exchange. To send the request, AsyncRequestDirect uses the Kernel primitive RequestDirect. AsyncRequestDirect then returns to the top of the asynchronous system service wait loop. When the response arrives, the context is resumed at the next program statement.

This operation should be used instead of AsyncRequest when it is necessary to use RequestDirect rather than the Request primitive. (See the descriptions of Request and RequestDirect in this chapter.)

Procedural Interface

AsyncRequestDirect (exch, pRq): ercType

where

exch

is the specified system service exchange.

pRq

is the memory address of the request block.

Request Block

AsyncRequestDirect is an object module procedure in Async.lib.

This page intentionally left blank

AtFileInit (pbFileSpec, cbFileSpec, pBuf, sBuf): ercType

Description

AtFileInit is called once to open and initialize a file containing character strings separated by white space (spaces, tabs, carriage returns). Each string in the file then is processed by subsequent calls to AtFileNext.

AtFileInit and AtFileNext are commonly used to handle a file containing a file list to be used for some other type of operation. The Executive, for example, uses these calls when it sees a file specification that begins with the atfile character (@).

Procedural Interface

AtFileInit (pbFileSpec, cbFileSpec, pBuf, sBuf): ercType

where

pbFileSpec
cbFileSpec

describe a valid file specification. The specification string must not include the atfile character (@). If wildcard characters are used, the resulting string match must be unique, or a status code will be returned.

pBuf
sBuf

describe a word-aligned buffer of 800 bytes to be used in subsequent calls to AtFileNext.

Request Block

AtFileInit is an object module procedure.

This page intentionally left blank

AtFileNext (pBuf, pSdRet): ercType

Description

AtFileNext returns the next string contained in the file opened in a previous call to AtFileInit. (See AtFileInit.)

Strings are separated in the file by white space, (spaces, tabs, carriage returns). The maximum size for a string is 255 characters.

Character sequences that begin with percent-quote (%') are assumed to be comments and are ignored until a closing quote is read, or until the current end of line is reached.

AtFileNext is called repetitively to process the next consecutive string until there are no more strings to return. Then it returns status code 1 ("End of file").

Procedural Interface

AtFileNext (pBuf, pSdRet): ercType

where

pBuf

is the memory address of the buffer supplied in a previous call to AtFileInit.

pSdRet

is the starting memory address of a 6 byte memory block returned to the caller. The address of the string is contained in the first four bytes. The last two bytes contain the count of bytes in the string.

Request Block

AtFileFNext is an object module procedure.

This page intentionally left blank

Beep: ercType

Description

Beep activates a single beep if the calling program owns the keyboard. Otherwise, it activates two short beeps, alerting the user that its source is not the keyboard owner.

When the calling program builds a request block, Beep activates the number of beeps specified in the *ercRet* field. In this case, the first beep is followed by short beeps.

Procedural Interface

Beep: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	52

This page intentionally left blank

BuildAsyncRequest (arg0, arg1,...,argn−1, rcRequestCode): ercType

NOTE: The parameter list shown for this operation is only a template to be filled in for a specified request. The actual arguments provided in the call to BuildAsyncRequest are the request parameters.

Description

BuildAsyncRequest is called by an asynchronous system service to send a request to the exchange serving the request. BuildAsyncRequest is called with the same parameter list that would be provided by the synchronous procedural interface except that the request code is also added at the end of the list.

Procedural Interface

BuildAsyncRequest (arg0, arg1,...,argn−1, rcRequestCode): ercType

where

rcRequestCode

is the request code of the specified request.

arg0, arg1,...,argn−1

are the parameters for the specified request.

Request Block

BuildAsyncRequest is an object module procedure in Async.lib.

BuildAsyncRequestDirect (arg0, arg1,...,argn−1, rcRequestCode, exchTarget): ercType

NOTE: The parameter list shown for this operation is only a template to be filled in for a specified request. The actual arguments provided in the call to BuildAsyncRequestDirect are the request parameters.

Description

BuildAsyncRequestDirectDirect is called by an asynchronous system service to send a request to a specified exchange. The parameters to BuildAsyncRequestDirect are the same as those provided to BuildAsyncRequest except that BuildAsyncRequestDirect takes the target exchange as an additional parameter. (See the description of BuildAsyncRequest.)

Procedural Interface

BuildAsyncRequestDirect (arg0, arg1,...,argn−1, rcRequestCode, exchTarget): ercType

where

arg0, arg1,...,argn−1

are the parameters for the specified request.

rcRequestCode

is the request code of the specified request.

exchTarget

is the number of the target exchange.

Request Block

BuildAsyncRequest is an object module procedure in Async.lib.

BuildFileSpec (userNum, pbOutputSpec, pcbOutputSpec, pcbOutputSpecNoPW, cbOutputSpecMax, fDefaultPath, pbNode, cbNode, pbVol, cbVol, pbDir, cbDir,pbFile, cbFile, fDefaultPrefix, pbPassword, cbPassword, fCanonical, bMode): word

Description

BuildFileSpec builds a file specification from the *tokens* (node, volume, directory, file name, and password strings) passed to it by the caller. The file specification and its size are returned at the specified memory addresses. For each level of the specification (node, volume, and so forth), BuildFileSpec inserts the appropriate brackets.

With each level built, BuildFileSpec performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a value of 0 is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*. The meanings of the nonzero values returned are listed in Table 11-3.)

Say, for example, the caller elects to build a full file specification from the tokens

Local
Sys
Dir1
Foo
1234

For each token, the caller provides the memory address of the token and the token size.

BuildFileSpec returns the following file specification and its size at the addresses provided by the caller:

{Local}[Sys]<Dir1>Foo^1234

In addition, BuildFileSpec returns the specification size excluding the password count and caret, if any, at a specified address. This allows applications such as Document Designer to use the file specification for two purposes: one for displaying the file specification without the password, and the other, for opening files that require the password.

To build the file specification, one or more tokens can be provided by the caller. The levels returned in the output file specification are determined by the memory addresses the caller passes.

If the address of a token and its size are zero values, the level for that token is not added to the output file specification, unless the default path flag *fDefault* is set to TRUE by the caller. In this case, the default path token from the User Configuration Block is used for that level.

NOTE: The fDefault *flag setting does not apply to file name tokens.*

To build each level of the file specification, BuildFileSpec calls the following lower layer procedures:

BuildSpecFromNode
BuildSpecFromVol
BuildSpecFromDir
BuildSpecFromFile
BuildSpecFromPassword

If your application only requires building a subset of the levels (one or two, for example), your application can call any of these procedures individually. For details, see the descriptions of these procedures in this chapter. There is also a discussion on accessing the file building procedures at different levels in "File Management" in the *CTOS Operating System Concepts Manual.*

Caution: *If an application builds a new file specification that does not use existing tokens in the file-specification buffer, it must first clear this buffer. Otherwise, tokens from a previous BuildFileSpec operation may be included in the new file specification.*

Procedural Interface

BuildFileSpec (userNum, pbOutputSpec, pcbOutputSpec, pcbOutputSpecNoPW,cbOutputSpecMax, fDefaultPath, pbNode, cbNode, pbVol, cbVol, pbDir, cbDir,pbFile, cbFile, fDefaultPrefix, pbPassword, cbPassword,fCanonical, bMode): word

where

userNum

is the user number (word). The default is 0 for the caller.

pbOutputSpec
pcbOutputSpec

are the memory addresses at which the file specification to be built and its size, respectively, will be returned. The size includes all brackets and the password count, if any.

pcbOutputSpecNoPW

is the memory address at which the size of the file specification, excluding the password count and the caret, if any, will be returned.

cbOutputSpecMax

is the maximum size of the file specification to be returned. If the length of the file specification built exceeds this size, an error is returned.

fDefaultPath

is a flag with one of the following values:

Value	Meaning
TRUE	For any token addresses and token sizes with zero values (except file tokens), the corresponding default level is placed in the output file specification.
FALSE	For any token addresses and token sizes with zero values, the corresponding default level is not placed in the output file specification.

pbNode
cbNode

describe the node token or are zero. If *pbNode* and *cbNode* are zero, the node token is not used to build the output file specification unless *fDefaultPath* is set to TRUE. Otherwise, the node token described is used to build the node level in the output file specification.

pbVol
cbVol

same as the description of *pbNode* and *cbNode* but applies to the volume token.

pbDir
cbDir

same as the description of *pbNode* and *cbNode* but applies to the directory token.

pbFile
cbFile

describe the file name token.

fDefaultPrefix

is a flag with one of the following values:

Value	Meaning
TRUE	The default prefix is added to the file name when the file name is added to the output file specification.
FALSE	The default prefix is not added to the file name.

pbPassword
cbPassword

same as the description of *pbNode* and *cbNode* but applies to the password token.

fCanonical

is a flag with one of the following values:

Value	Meaning
TRUE	The canonical forms of the node and volume levels are built into the output file specification.
FALSE	The values in the toeksn

bMode

is a byte value reserved for future use. Specify 0 for this parameter.

Request Block

BuildFileSpec is an object module procedure.

BuildFullSpecFromPartial (userNum, pbOutputSpec, pcbOutputSpec, pcbOutputSpecNoPW, cbOutputSpecMax, pbInputSpec, cbInputSpec, fDefaultPrefix, fCanonical, bMode): word

Description

BuildFullSpecFromPartial returns the full file specification based on the partial specification passed by the caller. BuildFullSpecFromPartial adds the defaults from the User Configuration Block for any levels (node, volume, directory, or password) not included in the file specification. In addition, BuildFullSpecFromPartial returns the size of the specification excluding the password and caret, if any, at the specified address.

BuildFullSpecFromPartial performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a 0 value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*. The meanings of the nonzero values returned are listed in Table 11-3.)

Procedural Interface

BuildFullSpecFromPartial (userNum, pbOutputSpec, pcbOutputSpec, pcbOutputSpecNoPW, cbOutputSpecMax, pbInputSpec, cbInputSpec, fDefaultPrefix, fCanonical, bMode): word

where

userNum

is the user number (word). The default is 0 for the caller.

pbOutputSpec

is the memory address at which the full file specification is returned.

pcbOutputSpec

is the memory address at which the size of the file specification is returned.

pcbOutputSpecNoPW

is the memory address at which the size of the file specification excluding the password count, if any, is returned.

cbOutputSpecMax

is the maximum size of the file specification to be built. If the length of the file specification built exceeds this size, an error is returned.

pbInputSpec
cbInputSpec

describe the partial file specification. This input specification is merged with default levels to create the output file specification.

fDefaultPrefix

is a flag. If TRUE, BuildFullSpecFromPartial returns the file prefix (if present) with the file specification. If FALSE, it omits the file prefix in the returned file specification.

fCanonical

is a flag (byte) with one of the following values:

Value	Meaning
TRUE	The canonical forms of the node and volume levels are built into the output file specification.
FALSE	The canonical forms are not used.

bMode

is a byte reserved for future use. Specify 0 for the value of this parameter.

Request Block

BuildFullSpecFromPartial is an object module procedure.

This page intentionally left blank

BuildLdtSlot (sl, la, wLimit, bAccess): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

Description

BuildLdtSlot constructs a descriptor in the caller's Local Descriptor Table (LDT).

Procedural Interface

BuildLdtSlot (sl, la, wLimit, bAccess) : ercType

where

sl

is the selector that identifes the descriptor.

la

is the 4-byte linear address that the LDT selector will reference.

wLimit

is the limit of the LDT selector.

bAccess

is the access byte of the LDT selector.

Request Block

BuildLdtSlot is a system–common procedure.

This page intentionally left blank

BuildSpecFromDir (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbDir, cbDir, bMode): word

Description

BuildSpecFromDir adds the directory to the output file specification returned to the caller. BuildSpecFromVol uses the directory token provided by the caller and returns the file specification with the directory added in at the specified address.

BuildSpecFromDir performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a 0 value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*. The meanings of the nonzero values returned are listed in Table 11-3.)

The BuildFileSpec operation calls this procedure to add the directory level to a file specification. (For details, see the description of BuildFileSpec in this chapter.)

Procedural Interface

BuildSpecFromDir (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbDir, cbDir, bMode): word

where

pbOutputSpec

is the memory address at which the file specification with the directory added is returned.

pcbOutputSpec

is the memory address at which the size of the file specification is returned.

cbOutputSpecMax

is the maximum size of the file specification to be built. If the length of the file specification built exceeds this size, an error is returned.

pbDir
cbDir

describe the directory token, or are zero values. (These parameters would only be zero if BuildFileSpec calls this procedure. For details, see the corresponding parameter descriptions in the BuildFileSpec procedure.) If your application is calling BuildSpecFromDir directly and you want to add the default directory token to the file specification, your application must call GetUcb before calling this procedure to obtain the pointer to and size of the default directory. (For details, see the description of GetUcb in this chapter.)

bMode

is a byte reserved for future use. Specify 0 for the value of this parameter.

Request Block

BuildSpecFromDir is an object module procedure.

BuildSpecFromFile (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbFile, cbFile, pbDefaultPrefix, cbDefaultPrefix, bMode): word

Description

BuildSpecFromFile adds a file name to the output file specification returned to the caller. BuildSpecFromFile uses the file name token provided by the caller and returns the file specification with the file name added at the specified address.

BuildSpecFromFile performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a 0 value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual.* The meanings of the nonzero values returned are listed in Table 11-3.)

The BuildFileSpec operation calls this procedure to add the file name level to a file specification. (For details, see the description of BuildFileSpec in this chapter.)

Procedural Interface

BuildSpecFromFile (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbFile, cbFile, pbDefaultPrefix, cbDefaultPrefix, bMode): word;

where

pbOutputSpec

is the memory address at which the file specification with the file name added is returned.

pcbOutputSpec

is the memory address at which the size of the file specification is returned.

cbOutputSpecMax

is the maximum size of the file specification to be built. If the length of the file specification built exceeds this size, an error is returned.

pbFile
cbFile

describe the file name token.

pbDefaultPrefix
cbDefaultPrefix

describe the default file prefix. If these parameters are zero values, the default prefix is not merged with the file name.

bMode

is a byte reserved for future use. Specify 0 for the value of this parameter.

Request Block

BuildSpecFromFile is an object module procedure.

BuildSpecFromNode (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbNode, cbNode, bMode): word

Description

BuildSpecFromNode adds the node to the output file specification returned to the caller. BuildSpecFromNode uses the node token provided by the caller and returns the file specification with the node added in at the specified address.

BuildSpecFromNode performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a 0 value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*. The meanings of the nonzero values returned are listed in Table 11-3.)

The BuildFileSpec operation calls this procedure to add the node level to a file specification. (For details, see the description of BuildFileSpec in this chapter.)

Procedural Interface

BuildSpecFromNode (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbNode, cbNode, bMode): word

where

pbOutputSpec

is the memory address at which the file specification with the node added is returned.

pcbOutputSpec

is the memory address at which the size of the file specification is returned.

cbOutputSpecMax

is the maximum size of the file specification to be built. If the length of the file specification built exceeds this size, an error is returned.

pbNode
cbNode

describe the node token, or are zero values. (These parameters would only be zero if BuildFileSpec calls this procedure. For details, see the corresponding parameter descriptions in the BuildFileSpec procedure.) If your application is calling BuildSpecForNode directly and you want to add the default node token to the file specification, your application must call GetUcb before calling this procedure to obtain the pointer to and size of the default node. (For details, see the description of GetUcb in this chapter.)

bMode

is a byte reserved for future use. Specify 0 for the value of this parameter.

Request Block

BuildSpecFromNode is an object module procedure.

BuildSpecFromPassword (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbPassword, cbPassword, bMode): word

Description

BuildSpecFromPassword adds a password to the output file specification returned to the caller. BuildSpecFromPassword uses the password token provided by the caller and returns the file specification with the password added at the specified address.

Unlike the other file building procedures called by BuildFileSpec (namely, BuildSpecFromNode, BuildSpecFromVol, BuildSpecFromDir, and BuildSpecFromFile), BuildSpecFromPassword requires that the caller pass the memory address of the exact position in the output file specification where the password is to be appended. This is because the password can be placed at the volume, directory, or file level of a file specification.

Say, for example, the caller provides the password token 1234 and the memory address of the file specification {Local}[Sys]<Dir1>. BuildSpecFromPassword adds the password at the directory level, as shown below:

{Local}[Sys]<Dir1>^1234

Like the other file building procedures, BuildSpecFromDir performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a 0 value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*. The meanings of the nonzero values returned are listed in Table 11-3.)

The BuildFileSpec operation calls this procedure to add the password level to a file specification. (For details, see the description of BuildFileSpec in this chapter.)

Procedural Interface

BuildSpecFromPassword (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbPassword, cbPassword, bMode): word

where

pbOutputSpec

is the memory address at which the file specification with the password added is returned.

pcbOutputSpec

is the memory address at which the size of the file specification is returned.

cbOutputSpecMax

is the maximum size of the file specification to be built. If the length of the file specification built exceeds this size, an error is returned.

pbPassword
cbPassword

describe the password token, or are zero values. (These parameters would only be zero if BuildFileSpec calls this procedure. For details, see the corresponding parameter descriptions in the BuildFileSpec procedure.) If your application is calling BuildSpecForPassword directly and you want to add the default password token to the file specification, your application must call GetUcb before calling this procedure to obtain the pointer to and size of the default password. (For details, see the description of GetUcb in this chapter.)

bMode

is a byte reserved for future use. Specify 0 for the value of this parameter.

Request Block

BuildSpecFromPassword is an object module procedure.

This page intentionally left blank

BuildSpecFromVol (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbVol, cbVol, bMode): word

Description

BuildSpecFromVol adds the volume to the output file specification returned to the caller. BuildSpecFromVol uses the volume token provided by the caller and returns the file specification with the volume added in at the specified address.

BuildSpecFromVol performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a 0 value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*. The meanings of the nonzero values returned are listed in Table 11-3.)

The BuildFileSpec operation calls this procedure to add the volume level to a file specification. (For details, see the description of BuildFileSpec in this chapter.)

Procedural Interface

BuildSpecFromVol (pbOutputSpec, pcbOutputSpec, cbOutputSpecMax, pbVol, cbVol, bMode): word

where

pbOutputSpec

is the memory address at which the file specification with the volume added is returned.

pcbOutputSpec

is the memory address at which the size of the file specification is returned.

cbOutputSpecMax

is the maximum size of the file specification to be built. If the length of the file specification built exceeds this size, an error is returned.

pbVol
cbVol

describe the volume token, or are zero values. (These parameters would only be zero if BuildFileSpec calls this procedure. For details, see the corresponding parameter descriptions in the BuildFileSpec procedure.) If your application is calling BuildSpecFromVol directly and you want to add the default node token to the file specification, your application must call GetUcb before calling this procedure to obtain the pointer to and size of the default volume. (For details, see the description of GetUcb in this chapter.)

bMode

is a byte reserved for future use. Specify 0 for the value of this parameter.

Request Block

BuildSpecFromVol is an object module procedure.

CallRealCommIsr

NOTE: This operation does not return a status code.

Description

CallRealCommISR transfers control to a real mode RS-232 communications raw interrupt handler (CRIH).

Procedural Interface

CallRealCommIsr

Request Block

CallRealCommIsr is a system-common procedure.

This page intentionally left blank

CacheClose(cachePoolHandle)

Description

CacheClose is called to invalidate the cache pool handle when the entire cache pool is no longer needed. After this call, the memory allocated for this cache pool can be returned to the system. This operation is useful for deinstallable services using the cache system-common procedures. Note that any necessary flushing must have been performed using the CacheFlush operation before calling this operation. If there are any dirty or unreleased entries in the pool when this operation is called, status code 1415 ("Cache pool busy") is returned.

Caution: It is the responsibility of the caller to return the memory space used by the cache pool to the operating system.

Procedural Interface

CacheClose(cachePoolHandle)

where

cachePoolHandle

is the cache pool handle (double word) for this cache pool returned by the CacheInit operation.

Request Block

CacheClose is a system-common procedure.

This page intentionally left blank

CacheFlush (cachePoolHandle, pCacheIdMask, pCacheIdTemplate, wControlMask, wControlWord, pcSelectedRet): ercType

Description

CacheFlush permits any subset of unlocked entries in a cache pool to be flushed. The selection criteria for flushing is based on the cache ID mask and cache ID template specified.

The state of each cache entry selected is altered according to the *wControlMask* and *wControlWord* parameters *if* the selected cache entry is unlocked. If, however, a selected entry is locked, its state is not changed and status code 1402 ("Cache entry locked") is returned. The status code indicates that at least one of the cache entries selected was locked and therefore not flushed, but all unlocked entries were processed.

This constraint may be circumvented by setting the flag bit *mCacheOverrideLock* in the control mask and control word. *mCacheOverrideLock* may be asserted only if the control mask and control word also specify that the cache entry be invalidated. By asserting *mCacheOverrideLock*, the caller is promising that any locked entries selected cannot possibly be active in the hands of another cache client. This bit allows a remote cache system service a means of terminating outstanding busy cache buffers.

Procedural Interface

CacheFlush(cachePoolHandle, pCacheIdMask, pCacheIdTemplate, wControlMask, wControlWord, pcSelectedRet): ercType

where

cachePoolHandle

is the cache pool handle (double word) returned by the CacheInit operation.

pCacheIdMask

is the memory address of an 8 byte mask that identifies which corresponding bits in *cacheIdTemplate* are to be used as a key to select cache entries. If a bit is set (is 1), this means the corresponding bit position in *cacheIdTemplate* is to be used in the key.

pCacheIdTemplate

is the memory address of an 8 byte template that specifies the key to be used in identifying the cache entries to be flushed. A cache entry is selected if the corresponding bit(s) in its ID, And'd with *cacheIdMask*, exactly match those in the template.

wControlMask

is a word that identifies the operational bits in *wControlWord*. If a bit is set (is 1), the corresponding bit in *wControlWord* is operational.

wControlWord

is a word of boolean bits indicating special operation(s) on the state of this cache entry to be performed before the entry is flushed. If a bit is set, the state is asserted. Not every combination is allowed. The only bits considered active are the those for which a bit is set in the corresponding position in *wControlMask*.

The bit assignments and their meanings are as follows:

Bit	Field	Description
0	mCacheValid	Valid state. Because an entry must have a valid ID for CacheFlush to be called, asserting this state is meaningless. This bit should only be cleared, enabling the caller to erase the cache ID associated with this cache entry so that the ID no longer exists in the pool.
2	mCacheDirty	Dirty state. This state indicates that a cache entry's data has been modified and has not been written to secondary storage. Dirty may only be asserted if the cache pool is asynchronous. (Because write-through caches always keep secondary storage in synchonization with the data in the cache, asserting this state is not allowed and results in the return of status code 1417 ("Bad cache control word"). Dirty should be asserted by an asynchronous cache user when a cache entry fetch has been made to modify data. It should be deasserted as part of the user-provided flushing procedure after the data is committed to storage. (For details on the flushing procedure, see *pFlushProc* in the CachInit operation description.)
3	mCacheMRU	Most-recently-used (MRU). If asserted (default), the cache entry is placed at the end of the LRU list (made most recently used) after being unlocked. If deasserted, the cache entry is placed at the beginning of the LRU list and becomes the next entry to be replaced by CacheGetEntry when a miss occurs.

Bit	Field	Description
4	mCacheOverrideLock	Overriding a lock. If the valid state is asserted, asserting override lock allows a locked entry to be flushed and its status altered.

NOTE: The mCacheOverrideLock *flag bit should be used with caution.*

pcSelectedRet

is the memory address of the count of entries (word) selected to be flushed by this operation.

Request Block

CacheFlush is a system-common procedure.

CacheGetEntry (cachePoolHandle, pCacheId, wFlags, ppCacheBufferRet, pCacheEntryHandleRet): ercType

Description

CacheGetEntry queries a cache pool for a specified cache entry. The operation first tries to find an entry in the cache that has the specified cache ID. If CacheGetEntry cannot find the entry and there are available entries on the LRU list, it creates a new entry by removing an entry from the LRU list. If the entry on the LRU list of an asynchronous cache pool is dirty, the cache pool's flushing procedure as specified to *CacheInit* is called. (For details on the flushing procedure, see the description of the *pFlushProc* parameter to CacheInit.)

The value of the status code returned by this operation establishes the meaning of the cache data buffer address (*pCacheBufferRet*) and the cache entry handle (*pCacheEntryHandleRet*) returned to the caller. From the code returned, the caller can determine the next course of action. For example, if status code 1403 is returned, the cache entry was not found in the cache and no cache entries are currently available. In such a case, the caller must provide the code necessary to coordinate waiting processes. Details on all the codes returned by this operation are included in the parameter description of *ppCacheBufferRet*. If this operation finds the specified cache entry in the cache pool, or if there is an available cache entry buffer on the LRU list, a cache entry handle is returned to the caller at the specified address.

Procedural Interface

CacheGetEntry (cachePoolHandle, pCacheId, wFlags, ppCacheBufferRet, pCacheEntryHandleRet): ercType

where

cachePoolHandle

is the cache pool handle (double word) returned by the CacheInit operation.

pCacheId

is the memory address of an 8 byte cache entry identification number (ID) that uniquely identifies the cache entry to be queried. The contents of the cache entry ID are under the control of the caller. Care must be taken to ensure that the ID is unique, however. An ID for disk caching performed by file management, for example, is uniquely identified by the device index and the virtual disk address (VDA) of the disk block(s).

wFlags

is a word containing flag bits that can be used to modify the processing performed by *CacheGetEntry*. The meanings of the bits are as follows:

Bit	Field	Description
0	mCacheDontSteal	If a cache entry with the specified cache ID does not currently exist in the cache pool (a cache miss), a new cache entry is not created. Status code 1411 ("No such entry") is returned.

Bit	Field	Description
1	mCacheDeferHit	If a cache entry with the specified cache ID occurs in the cache (a cache hit), rather than returning the locked cache entry to the caller, the cache entry is placed at the end of the LRU list and status code 1404 ("Cache hit deferred") is returned. The implication is that the caller is filling chained buffers (described below) and wants to pick up a hit encountered along the way later.
3	mCacheChained	The caller has previously obtained a cache entry, the handle for which is located at the memory address *pCacheEntryHandleRet*. If an entry is obtained with this call to CacheGetEntry, the new entry will be chained to the previous entry using the LRU link fields in the Cache Entry Descriptor. If the previous entry has no free thread (normal unchained case), it will be initialized and marked as the head of the chain. With respect to the previous entry, the new entry is the next in the chain. *pCacheEntryHandleRet* may be treated by the caller as the memory address of a cache chain element (with the format shown below). The cache entry handle and cache data buffer address are returned for a chained entry just as they are for nonchained entries.

A cache chain element has the following format:

Offset	Field	Size (bytes)	Description
0	pNext	4	Memory address of the next element in the cache chain.
4	pPrev	4	Memory address of the previous element in the cache chain.
8	pCacheBuffer	4	Memory address of the cache data buffer associated with this entry.

NOTE: A nonzero status code returned by this operation does not always mean that the operation is unsuccessful.

ppCacheBufferRet

is a pointer to the memory address at which the appropriate cache entry data buffer is returned. The status code returned by this operation determines the meaning of *pCacheBufferRet*. A nonzero value does not necessarily mean the operation was unsuccessful. The following codes are not truly errors but denote the conditions described:

Code	Meaning
0	("Erc OK"). The cache entry with the given ID was found (that is, this was a hit). The entry is now locked for the exclusive use of the caller until released by a call to CacheReleaseEntry. The cache entry handle returned describes the entry whose buffer address was returned.

Code	Meaning
1401	("Cache miss"). The cache entry with the given ID was not found (that is, this was a miss), but there is a cache entry buffer available for use (not locked for the exclusive use of other callers) in the cache pool. The address of the least-recently-used entry on the LRU list is returned, and the entry is marked with the caller's ID (locked for the exclusive use of the caller). Because the entry handle returned describes the entry from previous use (new cache ID has replaced a previously existing one), the buffer contents are invalidated, and the buffer is now ready to be filled by the caller. For example, the disk caching service would schedule the disk driver to transfer data into this cache entry.
1402	("Entry locked"). The cache entry was found, but it is currently locked out to another user. The returned pointer and the cache entry handle in this case are not valid. The calling program must implement code to handle this situation. For example, if the cache users are cooperating processes that queue up entries, the wait queue can be checked when a call to CacheReleaseEntry is made.
1403	("No free entry"). The cache entry was not found, and there are no available cache entry buffers in the LRU list. (Note that entries locked for exclusive use do not appear on the LRU list.) The returned pointer and the cache entry handle in this case are not valid.
1404	("Hit deferred"). An existing cache entry was found but the caller has asserted *mCacheDeferHit*. No cache entry is obtained, but the entry is placed at the end of the LRU list.
1411	("No such entry"). An existing cache entry was not found and the caller has asserted *mCacheDontSteal*. No cache entry is obtained.

In addition, this operation may return status codes that truly do mean errors, These include

Code	Meaning
1410	("Bad handle"). The supplied cache pool handle is invalid.
1412	("Bad descriptor"). The caller asserted *mCacheChained* but the cache entry handle at *pCacheEntryHandleRet* is invalid.

NOTE: It is the responsibility of the caller to synchronize external processes to accomodate the return of status codes 1402 and 1403.

pCacheEntryHandleRet

is the memory address of a double word to which the cache entry handle is returned if a cache entry is obtained (the operation results in a cache hit or miss). Although this double word is actually the memory address of the Cache Entry Descriptor, the caller should treat it as a handle of unknown internal format. The handle is used in a subsequent call to the CacheReleaseEntry operation to release the cache entry for reuse.

Request Block

CacheGetEntry is a system-common procedure.

CacheGetStatistics (cachePoolHandle, ppCsbRet): ercType

Description

CacheGetStatistics returns a pointer to the statistics for the specified cache at the memory address provided. The statistics are returned in the format of a Cache Statistics Block. The caller must treat the information returned by this operation as read only.

Procedural Interface

CacheGetStatistics(cachePoolHandle, ppCsbRet): ercType

where

cachePoolHandle

is the cache pool handle (double word) returned by the CacheInit operation.

ppCsbRet

is a pointer to the memory address where the Cache Statistics Block is returned. The format of the Cache Statistics Block is described below:

Offset	Field	Size (bytes)	Description
0	qsCacheSpace	4	Total amount of memory allocated to create the cache.
4	sBlock	2	Size of the data buffer.

Offset	Field	Size (bytes)	Description
6	cBlock	2	Number of data buffers in the cache pool. The amount of allocated memory lost to overhead for creating the cache data structures and for meeting buffer alignment requirements is *qsCacheSpace–(cBlock * sBlock)*.
8	cMinWorkingSet	2	Minimum number of nonsticky entries (minimum working set) in the cache pool. If the minimum working set falls below this value, status code 1414 ("Cache memory exhausted") is returned.
10	cValid	2	Number of valid cache entries (entries with defined cache IDs). *cBlock–cValid* indicates the number of invalid entries .
12	cFree	2	Number of entries in the LRU list. *cFree* includes two types of available entries: invalid entries (located towards the beginning of the list) and valid entries (located towards the end).
14	cLocked	2	Number of locked cache entries. An entry is locked when it is obtained by calling CacheGetEntry, and unlocked by calling CacheReleaseEntry. While an entry is locked, the entry is taken off the LRU list.
16	cSticky	2	Number of sticky entries. While an entry is sticky, the entry is taken off the LRU list and is not subject to removal from the cache.
18	cDirty	2	Number of locked cache entries that are dirty. *cDirty* only applies to a write-back (asynchronous) cache (the Cache Pool Descriptor of which must contain the address of a flushing procedure.) *cDirty* is always 0 for a write-through (synchronous) cache.

Offset	Field	Size (bytes)	Description
20	qcGets	4	Number of calls to GetCacheEntry.
24	qcHits	4	Number of GetCacheEntry calls that returned status code 0 ("ercOK"). (See the description of CacheGetEntry for details.)
28	qcMisses	4	Number of GetCacheEntry calls that returned status code 1401 ("Cache miss"). (See the description of CacheGetEntry for details.)
32	qcWaits	4	Number of CacheGetEntry calls that returned status code 1402 ("No cache buffer available") or status code 1403 ("Cache buffer locked"). It is the caller's responsibility to wait for the release of a cache buffer when either of these codes are returned. (See the description of CacheGetEntry for details.)

Request Block

CacheGetStatistics is a system-common procedure.

This page intentionally left blank

CacheGetStatus(cachePoolHandle, pCacheID, pControlWordRet): ercType

NOTE: A nonzero status code returned by this operation does not always mean that the operation is unsuccessful. A cache entry can be locked out to another user, for example, in which case status code 1402 is returned. (See the CTOS Status Codes Reference Manual *for details.)*

Description

CacheGetStatus returns status information for the specified cache entry.

Procedural Interface

CacheGetStatus(cachePoolHandle, pCacheID, pControlWordRet): ercType

where

cachePoolHandle

is the cache pool handle (double word) returned by the CacheInit operation.

pCacheId

is the memory address of an 8 byte string that uniquely identifies the cache entry whose status is to be queried. (For details, see the description of *pCacheId* in the CacheGetEntry operation.) Status information can only be returned if the cache entry currently exists.

pControlWordRet

is the memory address of a word containing the appropriate status information. The bits and their meanings are described below:

Bit	Field	Description
0	mCacheValid	The cache entry is valid.
1	mCacheDirty	The cache entry is dirty.
2	mCacheSticky	The cache entry is sticky.
3	mCacheChained	The cache entry is chained.

On a successful return of status, the flag bit *mCacheValid* will always be set.

Request Block

CacheGetStatus is a system-common procedure.

CacheInit (pCacheMemoryTable, sCacheMemoryTable, sDataBuffer, pFlushProc, wDataAlignMask, cMinWorkingSet, pCachePoolHandleRet): ercType

Description

CacheInit initializes a cache pool from memory previously allocated and returns the cache pool handle. The handle is used to refer to the pool in subsequent calls to the cache system-common procedures. The number of cache entries allocated within the cache memory area is a function of the memory available to the cache pool and the specified size of a cache entry.

Procedural Interface

CacheInit (pCacheMemoryTable, sCacheMemoryTable, sDataBuffer, pFlushProc, wDataAlignMask, cMinWorkingSet, pCachePoolHandleRet): ercType

where

pCacheMemoryTable

is the memory address of a table of pb/cb pairs that describe the memory available to the cache system-common procedures. The cb can be any length. If a cb has a value of 0FFFFh, it is considered to be *exactly 64K* bytes (65,536).

sCacheMemoryTable

is the size (in bytes) of the cache memory table. Because a pb/cb pair is 6 bytes long, *sCacheMemoryTable* must be a multiple of 6. The maximum table length is 65532 bytes (highest multiple of 6 in 64K). If certain alignments are required (for example, if DMA is used to empty and fill the cache data buffers), each pb/cb pair must adhere to the alignment. This is required for *wDataAlignMask* (below) to function correctly.

sDataBuffer

is the size (word) of the data buffer used by each cache entry. *sDataBuffer* does not include the size of the Cache Entry Descriptor associated with the buffer.

pFlushProc

is the memory address of the procedure to flush a dirty cache entry (containing modified data not written to disk) or is 0. If the cache pool is a write-through (synchronous) cache, *pFlushProc* is 0. Otherwise, if the cache pool is a write-back (asynchronous) cache pool, it may have entries marked as dirty by the CacheReleaseEntry operation. In this case, *pFlushProc* is the entry point for the procedure to flush the buffer. At the time the flushing procedure is called, the buffer is locked for the exclusive use of the caller who filled it. The buffer must be explicitly unlocked using the CacheReleaseEntry operation when buffer flushing is complete.

For efficiency, the flushing procedure must be asynchronous and re-entrant. This enables the cacher to reuse the procedure without having to wait for a previous flushing operation to complete. Once a buffer is flushed, however, the cache entry with which it is associated must be released and its dirty status cleared using the CachReleaseEntry operation. For this reason, the flushing procedure must return to a synchronization point so it can call CacheReleaseEntry at the proper time.

The caller's flushing procedure must conform to the following interface:

FlushProc (cachePoolHandle, pCacheId, cacheEntryHandle, pDataBufferToFlush)

where

cachePoolHandle

is a cache pool handle (double word) previously obtained by a call to the CacheInit operation.

pCacheId

is the memory address of the 8 byte cache identification number (ID) of the buffer to flush. (See the description of *pCacheId* in the CacheGetEntry operation.)

cacheEntryHandle

is the cache entry handle (double word). (See the description of *pCacheEntryHandleRet* in the CacheGetEntry operation.)

pDataBufferToFlush

is the memory address of the data buffer to be flushed.

wDataAlignMask

is a word that specifies the data alignment requirements (if any) at the time the cache entry data buffers are allocated. Asserted bits (right justified) indicate the boundary at which buffers must be allocated. For example, a value of 0 means there is no alignment requirement, 3FFh means allocate on 1K byte boundaries, and 0FFFh allocates on 4K byte boundaries. For this to work correctly, each of the memory segments described by the pb/cb pairs in the cache memory table must be aligned on the required boundary.

cMinWorkingSet

is a word that specifies the minimum number of buffers that must be available in the LRU list at any given time. If the number of buffers allocated to the cache minus the number of sticky buffers falls below this value, status code 1414 ("Cache memory exhausted") is returned.

pCachePoolHandleRet

is the memory address of a double word to which the handle for this cache pool is returned.

Request Block

CacheInit is a system-common procedure.

CacheReleaseEntry(cachePoolHandle, cacheEntryHandle, wControlMask, wControlWord): ercType

Description

CacheReleaseEntry releases (unlocks) a cache entry obtained by a call to the CacheGetEntry operation (or passed to the caller's flushing procedure for writing to secondary storage) and places the entry at the end of the least-recently-used (LRU) list. The cache pool handle returned by CacheInit and the cache entry handle returned by CacheGetEntry are used to identify the cache and the entry within it to release. The cache entry handle must define an entry that is valid and locked. If the cache entry is chained (*mCacheChained* was asserted when the entry was obtained through a previous call to *CacheGetEntry*), all entries in the same chain are processed.

Since CacheGetEntry locked the cache buffer for the exclusive use of the caller, CacheReleaseEntry must be called as soon as exclusive access is no longer needed. With the disk cache, for example, a cache entry for a Read request is released as soon as the contents of the cache buffer are copied to the client space.

Through the *wControlMask* and *wControlWord* parameters, the caller can optionally define a new status for the cache entry being released. Status code 1414 ("Cache memory exhausted") is returned if making the cache entry sticky would cause the number of entries to fall below the minimum working set established when the cache was initialized with the CacheInit operation.

Unless explicitly indicated a released cache entry is not invalidated. Only when a particular entry has moved to the head of the LRU list and a call to CacheGetEntry is made that returns status code 1401 ("Cache miss") is the entry invalidated. (For details, see the description of CacheGetEntry.)

Procedural Interface

CacheReleaseEntry(cachePoolHandle, cacheEntryHandle, wControlMask, wControlWord): ercType

where

cachePoolHandle

is the cache pool handle (double word) returned by the CacheInit operation.

cacheEntryHandle

is the cache entry handle (double word) returned by the CacheGetEntry operation.

wControlMask

is a word that identifies the operational bits in *wControlWord*. If a bit is set (is 1), the corresponding bit in *wControlWord* is operational.

wControlWord

is a word of boolean bits indicating special operation(s) on the state of this cache entry to be performed before the entry is released. If a bit is set, the state is asserted. Not every combination is allowed. The only bits considered active are the those for which a bit is set in the corresponding position in *wControlMask*.

The bit assignments and their meanings are as follows:

Bit	Field	Description
0	mCacheValid	Valid state. Because an entry must have a valid ID for CacheReleaseEntry to be called, asserting this state is meaningless. This bit should only be cleared, enabling the caller to erase the cache ID associated with this cache entry so that the ID no longer exists in the pool.
1	mCacheDirty	Dirty state. This state indicates that a cache entry's data has been modified and has not been written to secondary storage. Dirty may only be asserted if the cache pool is asynchronous. (Because write-through caches always keep secondary storage in synchonization with the data in the cache, asserting this state is not allowed and results in the return of status code 1417 ("Bad cache control word"). Dirty should be asserted by an asynchronous cache user when a cache entry fetch has been made to modify data. It should be deasserted as part of the user-provided flushing procedure after the data is committed to storage. (For details on the flushing procedrue, see *pFlushProc* in the CachInit operation description.)

Bit	Field	Description
2	mCacheSticky	Sticky state. If this state is asserted, an entry remains in the cache after being unlocked and is not returned to the LRU list where it would be subject to removal. Setting and clearing this bit is allowed. Asserting this state keeps a cache entry in memory until the state is deasserted.
3	mCacheMRU	Most-recently-used (MRU). If asserted (default), the cache entry is placed at the end of the LRU list (made most recently used) after being unlocked. If deasserted, the cache entry is placed at the beginning of the LRU list and becomes the next entry to be replaced by CacheGetEntry when a miss occurs.

NOTE: The caller must manage the case of several clients waiting for this specific released cache entry (or any available entry). (For details, see the description of CacheGetEntry.)

Request Block

CacheReleaseEntry is a system-common procedure.

CdAbsoluteRead (fh, wSecCnt, dwSecStart, pbRqData, cbRqData, psDataRet): ercType

Description

CdAbsoluteRead reads the specified number of disc sectors from a CD-ROM device, and writes the sector contents to a caller-supplied memory area. The device must have been previously opened by the CdOpen operation. (See CdOpen.)

CdAbsoluteRead differs from CdRead in one respect: when an application uses CdRead, it must specify the disc logical file addresses from which data is read. However, when an application uses CdAbsoluteRead, it must specify the disc sectors from which data is read.

Status code 15710 ("Invalid return buffer") is returned if the buffer at *pbRqData* is too small. This status code is also returned if *pbRqData* is 0. Status code 15701 ("Invalid CD-ROM handle") is returned if the specified device handle is invalid.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdAbsoluteRead (fh, wSecCnt, dwSecStart, pbRqData, cbRqData, psDataRet): ercType

where

fh

is the file handle returned by the CdOpen operation.

wSecCnt

is the number of sectors to be read.

dwSecStart

is the starting sector to be read.

pbRqData
cbRqData

describe the memory area to which sector data is written.

psDataRet

is the memory address of the buffer to which the count of bytes successfully read is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	10
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33203
12	fh	4	
16	wSecCnt	2	
18	dwSecStart	4	
22	pbRqData	4	
26	cbRqData	2	
28	psDataRet	4	
32	ssDataRet	2	2

This page intentionally left blank

CdAudioCtl (fh, pbAudioParms, cbAudioParms, pbAudioInfo, cbAudioInfo): ercType

Description

CdAudioCtl selects CD-ROM audio playback channels and their volume settings, and controls audio playback, pause, and stop features. In addition, this operation provides information on the disc as a whole, on particular tracks, and on the current Q-channel address.

To specify the function to be performed, the application provides the function's number in the buffer at *pbAudioInfo*. The following is a list of the functions currently offered by CdAudioCtl:

Function	Number	Description
Audio Disk Info	1	Reads the Q subchannel in the lead-in track to obtain the highest and lowest track numbers, as well as the starting address of the lead-out track. The lead-out starting address follows the minute-second-frame (Red Book) format.
Audio Track Info	2	Returns the Red Book address for the starting point of a track, as well as control information for that track. The track number specified must fall within the range of tracks returned by the Audio Disk Info function.

Function	Number	Description
Audio Q-Channel Info	3	Returns the current Q-channel address. An application can use this function to monitor the address of the read head while an audio track is played.
Audio Channel Control Write	4	Selects the volume level and audio playback channel for the left and right headphones. Up to four channels of audio information may be recorded on a compact disc. Any two of those four possible channels can be routed to the two output channels of the disc drive headphone jack. Monophonic sound can be obtained by setting both channel input bytes to the same value.
Audio Channel Control Read	5	Returns the volume level and audio playback channel of the left and right headphones.
Audio Pause Play	6	Pauses the current Audio Play function. Audio Play may be resumed with the Audio Resume function. While in an audio pause condition, the CD–ROM drive remains in a busy state.
Audio Stop Play	7	Stops an Audio Play function, and causes the CD-ROM drive to enter a not-busy condition.

Function	Number	Description
Audio Resume Play	8	The Audio Resume Play function resumes an Audio Play operation that was paused by an Audio Pause Play function.
Audio Play	9	The Audio Play function starts an audio playback operation. Playback continues until the requested ending track has been reached, the requested Red Book address has been reached, or an Audio Stop function occurs.
Audio Play Status	10	The Audio Play Status function returns the status of the current Audio Play operation. Additionally, this function returns the playback head's current address in both LBA (Logical Block Address) and MSF (Minutes, Seconds, Frame) formats.

Note that the device must have been previously opened by the CdOpen operation. (See CdOpen.)

Status code 15714 ("Invalid function") is returned if an invalid function number is specified. Status code 15701 ("Invalid CD-ROM handle") is returned if *fh* is an invalid file handle. Status code 15710 ("Invalid return buffer") is returned if the buffer at *pbAudioInfo* is too small. This status code is also returned if *pbAudioInfo* is 0.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdAudioCtl (fh, pbAudioParms, cbAudioParms, pbAudioInfo, cbAudioInfo): ercType

where

fh

is the file handle returned by a previous call to CdOpen.

pbAudioParms
cbAudioParms

describe a buffer which contains parameters for the audio operation. The first byte in this buffer always identifies the type of operation to be performed.

pbAudioInfo
cbAudioInfo

describe the buffer to which the audio information is returned.

Audio Disk Info Function

To select the Audio Disk Info function, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 1 causes CdAudioCtl to perform an Audio Disk Info function.

The following information is then returned at *pbAudioInfo:*

Offset	Field	Size (Bytes)	Description
0	lowTrack	1	The lowest track number on the CD-ROM disc.
1	highTrack	1	The highest track number on the CD-ROM disc.
2	leadoutStart	4	The starting address of the lead-out track.

Audio Track Info Function

To select the Audio Track Info function, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 2 causes CdAudioCtl to perform a Track Info function.
1	trackNumber	1	The number of the track for which information is returned.

The following information is then returned at *pbAudioInfo:*

Offset	Field	Size (Bytes)	Description
0	minutes	1	The minute number at which the specified track begins.
1	seconds	1	The second number at which the specified track begins.
2	frame	1	The frame number at which the specified track begins.
3	controlAdr	1	A byte which contains track control information.

In the *controlAdr* field, each group of bit positions, when set or cleared, indicates the following:

Bit Positions	State	Description
Bits 0 to 3, 5	Cleared	Digital copy is not allowed.
Bits 0 to 4, 6, 7	Cleared	2 audio channels without pre-emphasis.
Bit 5	Set	2 audio channels with pre-emphasis.
Bit 8	Set	4 audio channels without pre-emphasis.
Bits 5 and 8	Set	4 audio channels with pre-emphasis.

Bit Positions	State	Description
Bit 7	Set	The track is a data track, not an audio track.
Bits 5 and 7	Set	Reserved.
Bit 6	Set	Digital copy is allowed.
Bits 7 and 8	Set	Reserved.

Audio Q-Channel Function

To select the Audio Q-Channel function, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 3 causes CdAudioCtl to perform an Audio Q-Channel function.

The following information is then returned at *pbAudioInfo:*

Offset	Field	Size (Bytes)	Description
0	controlAdr	1	See the Audio Track Info function for a description.
1	trackNo	1	The track number of the current head location.

Offset	Field	Size (Bytes)	Description
2	index	1	The index number in the current track number.
3	relativeMinute	1	The minute value, relative to the start of the track, of the current head location.
4	relativeSecond	1	The second value, relative to the start of the track, of the current head location.
5	relativeFrame	1	The frame value, relative to the start of the track, of the current head location.
6	absoluteMinute	1	The minute value, relative to the start of the disc, of the current head location.
7	absoluteSecond	1	The second value, relative to the start of the disc, of the current head location.
8	absoluteFrame	1	The frame value, relative to the start of the disc, of the current head location.

Audio Channel Control Write function

To select the Audio Channel Control Write function, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 4 causes CdAudioCtl to perform an Audio Channel Control Write function.
1	leftChanInput	1	The channel that provides sound to the left side of the headphone. This channel is usually 1.
2	leftChanVol	1	The volume level for the left channel. This value ranges from 0 (no sound) to 255 (maximum volume). The default value is 255.
3	rightChanInput	1	The channel that provides sound to the right side of the headphone. This channel is usually 2.
4	rightChanVol	1	The volume level for the right channel. This value ranges from 0 (no sound) to 255 (maximum volume). The default value is 255.

The Audio Channel Control Write function does not return information at *pbAudioInfo*.

Audio Channel Control Read function

To select the Audio Channel Control Read function, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 5 causes CdAudioCtl to perform an Audio Channel Control Read function.

The following information is then returned at *pbAudioInfo*:

Offset	Field	Size (Bytes)	Description
0	leftChanInput	1	The channel that provides sound to the left side of the headphone.
1	leftChanVol	1	The volume level for the left side of the headphone.
2	rightChanInput	1	The channel that provides sound to the right side of the headphone.
3	rightChanVol	1	The volume level for the right side of the headphone.

Audio Pause Play Function

To select the Audio Pause Play function, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 6 causes CdAudioCtl to perform an Audio Pause Play function.

The Audio Pause Play function does not return information at *pbAudioInfo*.

Audio Stop Play Function

To select the Audio Stop Play function, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 7 causes CdAudioCtl to perform an Audio Stop Play function.

The Audio Stop Play function does not return information at *pbAudioInfo*.

Audio Resume Play Function

To select the Audio Resume Play function, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 8 causes CdAudioCtl to perform an Audio Resume Play function.

The Audio Resume Play Function does not return information at *pbAudioInfo*.

Audio Play Function

To select the Audio Play function for the LBA format, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 9 causes CdAudioCtl to perform an Audio Play function.
1	wait	1	A value which indicates how long the caller must wait to regain control after a playback operation. If this value is 0, the caller will immediately regain control.
2	addressMode	1	A value that is always 0.

Offset	Field	Size (Bytes)	Description
3	startingTrack	1	The first track from which data is played back.
4	startingIndex	1	The first index from which data is played back.
5	endingTrack	1	The last track from which data is played back.
6	endingIndex	1	The last index from which data is played back.

To select the Audio Play function for the MSF format, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 9 causes CdAudioCtl to perform an Audio Play function.
1	wait	1	A value which indicates how long the caller must wait to regain control after a playback operation. If this value is 0, the caller will immediately regain control.
2	addressMode	1	The address mode. A value of 1 indicates that the values in the next fields are Red Book values (minutes, seconds, and frames).

Offset	Field	Size (Bytes)	Description
3	startingMinute	1	The first minute at which data is played back.
4	startingSecond	1	The first second at which data is played back.
5	startingFrame	1	The first frame at which data is played back.
6	endingMinute	1	The last minute at which data is played back.
7	endingSecond	1	The last second at which data is played back.
8	endingFrame	1	The last frame at which data is played back.

The Audio Play function does not return information at *pbAudioInfo*.

Audio Play Status Function

To select the Audio Play Status function, the caller provides a buffer at *pbAudioParms* with the following format:

Offset	Field	Size (Bytes)	Description
0	function	1	A value of 10 causes CdAudioCtl to perform an Audio Pause Status function.

The following information is then returned at *pbAudioInfo*:

Offset	Field	Size (Bytes)	Description
0	status	1	The status of the current Audio Play function.
1	currentAddressLBA	4	The current LBA (logical block address) at which the playback head is located.
5	currentAddressMin	1	The minute at which the playback head is located.
6	currentAddressSec	1	The second at which the playback head is located.
7	currentAdrsFrame	1	The frame at which the playback head is located.

The *status* field contains values that indicate the following:

Value	Description
0	The *status* byte is not supported.
1	The Audio Play function is in progress.
2	The Audio Play function is paused.
3	The Audio Play function completed successfully.
4	The Audio Play function stopped due to an error.
5	No status to report.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33205
12	fh	4	
16	reserved	2	
18	pbAudioParms	4	
22	cbAudioParms	2	
24	pbAudioInfo	4	
28	cbAudioInfo	2	

CdControl (fh, pbInfoRet, cbInfoRet, psInfoRet, wFunction): ercType

Description

CdControl controls the CD-ROM drive's disc eject and door locking mechanisms. It also returns information on the CD-ROM drive. The following describes the functions that CdControl performs:

Function	Description
Location of Head	Returns the current head location in both LBA (logical block address) and MSF (minutes, seconds, frames) formats.
Device Status	Returns the current device status.
Return Sector Size	Returns the device sector size.
Return Volume Size	Returns the size, in sectors, of the disc.
Eject Disc	Causes the disc to be ejected from the CD-ROM drive.
Lock Door	Prevents a person from manually ejecting the disc from the CD-ROM drive. The drive door remains locked until the application that locked the door performs an unlock operation, or until that application terminates.
Unlock Door	Reverses the lock condition and enables the eject button on the CD-ROM drive.

The application specifies the function to be performed, and provides a handle for the appropriate device. Note that the device must have been previously opened by the CdOpen operation. (See CdOpen.)

Status code 15710 ("Invalid return buffer") is returned if the buffer at *psInfoRet* is too small. This status code is also returned if *psInfoRet* is 0. Status code 15701 ("Invalid CD-ROM handle") is returned if the specified handle is invalid. Status code 15714 ("Invalid function") is returned if an invalid function number has been selected.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdControl (fh, pbInfoRet, cbInfoRet, psInfoRet, wFunction): ercType

where

fh

is the device handle returned by a previous call to the CdOpen operation.

pbInfoRet
cbInfoRet

describe the buffer to which data is returned. If CdControl performs a Location Of Head function, the returned data has the following format:

Offset	Field	Size (Bytes)	Description
0	headLocationLBA	4	The logical block address value of the current head location.
4	headLocMinutes	1	The minute value of the current head location.

Offset	Field	Size (Bytes)	Description
5	headLocSeconds	1	The second value of the current head location.
6	headlocFrame	1	The frame value of the current head location.

If CdControl performs a Device Status function, the returned data has the following format:

Bit Number	State	Description
0	0	The CD-ROM drive door is closed.
	1	The CD-ROM drive door is open.
1	0	The CD-ROM drive door is locked.
	1	The CD-ROM drive door is unlocked.
2	0	Only a portion of each sector can be read from the CD-ROM device.
	1	The entire contents of each sector can be read from the CD-ROM device.
3	0	The CD-ROM is a read-only device.
	1	The CD-ROM is a read/write device.
4	0	The CD-ROM device reads data but does not play audio/video tracks.
	1	The CD-ROM device reads data and plays audio/video tracks.

Bit Number	State	Description
5	0	The CD-ROM device does not support sector interleaving.
	1	The CD-ROM device supports sector interleaving.
6	0	Reserved
	1	Reserved
7	0	The CD-ROM device cannot pre-fetch data from the disc.
	1	The CD-ROM device can pre-fetch data from the disc.
8	0	The CD-ROM device does not allow control of the audio channels.
	1	The CD-ROM device allows control of the audio channels.
9	0	The CD-ROM device supports the LBA (Logical Block Address) addressing mode.
	1	The CD-ROM device supports the LBA (Logical Block Address) and Red Book addressing modes.
10–31		Reserved

If CdControl performs a Return Sector Size function, the returned data has the following format:

Offset	Field	Size (Bytes)	Description
0	readMode	1	The read mode. A value of 0 indicates that only a portion of each sector can be read from the CD-ROM device. A value of 1 indicates that the entire contents of each sector can be read.
1	sectorSize	2	The sector size for the device.

If CdControl performs a Return Volume Size function, the returned data has the following format:

Offset	Field	Size (Bytes)	Description
0	volumeSize	4	The size, in sectors, of the volume.

No information is returned if the Eject Disc, Lock Door, or Unlock Door functions are performed.

psInfoRet

is the memory address of the word containing the count of bytes returned.

wFunction

specifies the control function to be performed by the CD-ROM service. Each value indicates the following:

Value	Description
1	Location of Head function
2	Device Status function
3	Return Sector Size function
4	Return Volume Size function
5	Eject Disc function
6	Lock Door function
7	Unlock Door function

Request Block

ssInfoRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33206
12	fh	4	
16	wFunction	2	
18	pbInfoRet	4	
22	cbInfoRet	2	
24	psInfoRet	4	
28	ssInfoRet	2	2

This page intentionally left blank

CdClose (fh): ercType

Description

CdClose closes a CD-ROM file or device that was opened by a previous CdOpen operation. (See CdOpen.) The caller provides the handle of the file or device to be closed.

An application should use CdClose when it no longer requires access to the CD-ROM device. Once the CD-ROM device is closed, other applications can access the device.

Status code 15701 ("Invalid CD-ROM handle") is returned if the specified file handle is invalid.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdClose (fh): ercType

where

fh

is the file handle returned by a previous call to the CdOpen operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33213
12	fh	4	

CdDirectoryList (pbDevSpec, cbDevSpec, pbListRet, cbListRet, psDataRet): ercType

Description

CdDirectoryList returns a list which contains the path name of each directory on a CD-ROM disc. This operation can return directory lists for CD-ROM devices on nonlocal systems.

Status code 15702 ("Invalid specification") is returned if the device specification at *pbDevSpec* is invalid. Status code 15710 ("Invalid return buffer") is returned if the buffer at *pbListRet* is too small. This status code is also returned if *pbListRet* is 0.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdDirectoryList (pbDevSpec, cbDevSpec, pbListRet, cbListRet, psDataRet): ercType

where

pbDevSpec
cbDevSpec

describe a string which contains the device specification. This string has the form {Node}[DevName]. Square brackets are optional for the device name. CdDirectoryList does not distinguish between uppercase and lowercase characters.

pbListRet
cbListRet

describe the buffer to which the directory list is to be returned. Each directory list entry is a character string in which the first byte is the size of the string. The directories in a multiple-level path are separated by a backslash (\) character.

psDataRet

is the memory address of the word where the count of bytes in the directory list is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33217
12	reserved	6	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pbBuff	4	
28	cbBuff	2	
30	psDataRet	4	
34	ssDataRet	2	2

CdGetDirEntry (pbPathName, cbPathName, pbDirEntry, cbDirEntry, psDataRet): ercType

Description

CdGetDirEntry returns a CD-ROM directory record, in either ISO or High Sierra format, for a specified path. The caller provides the memory address where the path name resides. Note that the device must have been previously opened by the CdOpen operation. (See CdOpen.)

Status code 15710 ("Invalid return buffer") is returned if the buffer at *pbDirEntry* is too small. This status code is also returned if *pbDirEntry* is 0. Status code 15711 ("Device not ready") is returned if the CD-ROM device is not ready This condition typically occurs when the CD-ROM disc is not properly inserted in the device.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdGetDirEntry (pbPathName, cbPathName, pbDirEntry, cbDirEntry, psDataRet): ercType

where

pbPathName
cbPathName

describe the memory area where the path name resides.

pbDirEntry
cbDirEntry

describe the buffer to which the the directory record is to be returned. (For a description of the directory record, see "CD-ROM Service" in the *CTOS Programming Guide*.)

psDataRet

is the memory address to which the count of bytes returned is written.

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33207
12	reserved	6	
18	pbPathName	4	
22	cbPathName	2	
24	pbDirEntry	4	
28	cbDirEntry	2	
30	psDataRet	4	
34	ssDataRet	2	2

CdGetVolumeInfo (fh, pBufferRet, sBufferMax, psDataRet, wFunction):
ercType

Description

CdGetVolumeInfo returns the name of the copyright, abstract, or bibliographic file on a CD-ROM disc. In addition, it can return the disc's primary volume descriptor. The caller provides the handle of the CD-ROM device for which disc information is to be returned.

Note that the device must have been previously opened by the CdOpen operation. (See CdOpen.)

Status code 15710 ("Invalid return buffer") is returned if the buffer at *pbDirEntry* is too small. This status code is also returned if *pbDirEntry* is 0. Status code 15714 ("Invalid function") is returned if *wFunction* is not a permitted function number.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide.*) The service must be installed for this operation to be used.

Procedural Interface

CdGetVolumeInfo (fh, pBufferRet, sBufferMax, psDataRet, wFunction):
ercType

where

fh

is the device handle returned by a previous call to the CdOpen operation.

pBufferRet
sBufferMax

describe the memory area to which data is returned.

psDataRet

is the memory address to which the count of bytes returned is written.

wFunction

specifies which function CdGetVolumeInfo is to perform. Each function retrieves the following information:

Function Number	Information Retrieved
1	Copyright filename ISO copyright filename is 37 bytes High Sierra copyright filename is 32 bytes
2	Abstract filename (requires 38 bytes) ISO abstract filename is 37 bytes High Sierra abstract filename is 32 bytes
3	Bibliographic file name (requires 38 bytes) ISO bibliographic file name is 37 bytes High Sierra bibliographic file name is 32 bytes
4	Primary volume descriptor File format portion of descriptor is 14 bytes Entire descriptor is 2048 bytes

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33208
12	fh	4	
16	wFunction	2	
18	pBufferRet	4	
22	sBufferRet	2	
24	psDataRet	4	
28	ssDataRet	2	2

This page intentionally left blank

CdOpen (mode, pbFileSpec, cbFileSpec, pbPassword, cbPassword, pFh): ercType

Description

CdOpen opens a CD-ROM file or device on either a local or a non-local system and returns a file or device handle. Subsequent operations use this handle to refer to the file or device.

After the application has opened and accessed the CD-ROM device, it can close the device by calling CdClose. (See CdClose.)

Status code 15702 ("Invalid specification") is returned if the file or device specification at *pbFileSpec* is invalid. Status code 15705 ("Device in use") is returned if another user number is currently using the CD-ROM device in private mode.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdOpen (mode, pbFileSpec, cbFileSpec, pbPassword, cbPassword, pFh): ercType

where

mode

indicates the mode in which the CD-ROM file is to be opened. Each value indicates the following:

Value	Description
0	Share mode
1	Private mode

pbFileSpec
cbFileSpec

describe the file or device specification of the file or device to be opened. Valid file specifications include CTOS file specifications of the form

{Node}[CDROMn]<Directory>FileName

or ISO hierarchical file specifications of the form

{Node}[CDROMn]\Directory\Subdirectory\FileName.

CdOpen does not check for uppercase or lowercase letters when it matches file names.

pbPassword
cbPassword

describe the volume, directory, or file password that authorizes access to the specified file.

pFh

is the memory address of a quad to which the file or device handle is returned.

Request Block

sFhMax is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33212
12	reserved	2	
14	mode	2	
16	reserved	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pFh	4	
34	sFhMax	2	4

This page intentionally left blank

CdRead (fh, lfa, pBufferRet, sBufferMax, psDataRet): ercType

Description

CdRead transfers a specified number of bytes from a file on a CD-ROM device to a caller-specified memory area. The device must have been previously opened by the CdOpen operation. (See CdOpen.)

CdRead differs from CdAbsoluteRead in one respect: when an application uses CdAbsoluteRead, it must specify the disc sectors from which data is read. However, when an application uses CdRead, it must specify the disc logical file addresses from which data is read.

Status code 15709 ("End of file") is returned if the end of the file is reached before all the bytes are returned. Status code 15720 is returned if *lfa* is not a multiple of 512.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdRead (fh, lfa, pBufferRet, sBufferMax, psDataRet): ercType

where

fh

is the file handle returned by a previous CdOpen operation.

lfa

is the byte offset, from the beginning of the file, of the first byte to be read. This offset must be a multiple of 512.

pBufferRet

is the memory address of the buffer to which the data is returned. The buffer must be word aligned.

sBufferMax

is the number of bytes to be read from the CD-ROM device. This number must be a multiple of 512.

psDataRet

is the memory address of the word where the count of bytes successfully read is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33209
12	fh	4	
16	lfa	4	
20	pBufferRet	4	
24	sBufferMax	2	
26	psDataRet	4	
30	ssDataRet	2	2

CdSearchClose (sh): ercType

Description

CdSearchClose closes a CD-ROM search session that was previously opened by a CdSearchFirst operation. (See CdSearchFirst.) The caller provides CdSearchClose with the search handle of the session to be closed. Note that the device must have been previously opened by the CdOpen operation. (See CdOpen.)

Status code 15701 ("Invalid CD-ROM handle") is returned if the specified search handle is invalid.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdSearchClose (sh): ercType

where

sh

is the search handle returned by a previous call to the CdSearchFirst operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33214
12	sh	4	

CdSearchFirst (pbFileSpec, cbFileSpec, pbPassword, cbPassword, pbSearchBuf, cbSearchBuf, pbPathInfo, cbPathInfo, pSh): ercType

Description

CdSearchFirst opens a search session and returns the directory record and path of the first file on a CD-ROM disc which matches the caller-specified file name. The device must have been previously opened by the CdOpen operation. (See CdOpen.)

For wild-card file specifications, CdSearchFirst only returns information for the first matching file. To obtain information on remaining files that match the file specification, an application may repeatedly call the CdSearchNext operation (see CdSearchNext.)

When the application does not require any further information, it may close the search session by calling CdSearchClose. (See CdSearchClose.)

Status code 15702 ("Invalid specification") is returned if the file specification at *pbFileSpec* is invalid. Status code 15715 ("No such CD-ROM file") is returned if the specified file is not found.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdSearchFirst (pbFileSpec, cbFileSpec, pbPassword, cbPassword, pbSearchBuf, cbSearchBuf, pbPathInfo, cbPathInfo, pSh): ercType

where

pbFileSpec
cbFileSpec

describe the file specification of the file for which information is to be returned. Valid file specifications include CTOS file specifications of the form {Node}[CDROMn]<Directory>FileName, or ISO hierarchical file specifications of the form {Node}[CDROMn]\Directory\Subdirectory\FileName. CdSearchFirst does not check for uppercase or lowercase letters when it matches file names.

pbPassword
cbPassword

describe the volume, directory, or file password that authorizes access to the specified file.

pbSearchBuf
cbSearchBuf

describe the buffer to which the directory record is to be returned.

pbPathInfo
cbPathInfo

describe the buffer to which the file's path is to be returned.

pSh

is the memory address of a quad to which the search handle is returned.

Request Block

sShMax is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33215
12	reserved	2	
14	reserved	2	
16	reserved	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pSh	4	
34	sShMax	2	4
36	pbSearchBuf	4	
40	cbSearchBuf	2	
42	pbPathInfo	4	
46	cbPathInfo	2	

This page intentionally left blank

CdSearchNext (sh, pbSearchBuf, cbSearchBuf, pbPathInfo, cbPathInfo): ercType

Description

CdSearchNext returns the directory record and path of the next CD-ROM file that matches a wild-card file specification previously scanned by the CdSearchFirst operation. (See CdSearchFirst.) The CD-ROM device must have been previously opened by the CdOpen operation. (See CdOpen.)

To obtain information on all matching files, an application may repeatedly call CdSearchNext until this operation returns the status code 15715 ("No such CD-ROM file").

The application provides the search handle previously returned by a CdSearchFirst operation, as well as the memory addresses of buffers where the file information is to be returned.

Status code 15710 ("Invalid return buffer") is returned if the buffer at either *pbSearchBuf* or *pbPathInfo* is too small. This status code is also returned if either *pbSearchBuf* or *pbPathInfo* is 0.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdSearchNext (sh, pbSearchBuf, cbSearchBuf, pbPathInfo, cbPathInfo): ercType

where

sh

is the search handle returned by a previous call to the CdSearchFirst operation.

pbSearchBuf
cbSearchBuf

describe the buffer to which directory record information is to be returned.

pbPathInfo
cbPathInfo

describe the buffer to which the file path is to be returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33216
12	sh	4	
16	reserved	2	
18	pbSearchBuf	4	
22	cbSearchBuf	2	
24	pbPathInfo	4	
28	cbPathInfo	2	

This page intentionally left blank

CdServiceControl (pbDevSpec, cbDevSpec, pbRqData, cbRqData, pbBuffer, cbBuffer, psDataRet, wFunction): ercType

NOTE: CdServiceControl can only deinstall a CD-ROM service that is installed on a local system.

Description

CdServiceControl deinstalls the CD-ROM Service that controls the specified device on the local workstation.

Status code 15702 ("Invalid specification") is returned if the device specification at *pbDevSpec* is invalid.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdServiceControl (pbDevSpec, cbDevSpec, pbRqData, cbRqData, pbBuffer, cbBuffer, psDataRet, wFunction): ercType

where

pbDevSpec
cbDevSpec

describe a character string of the form [DevName]. Square brackets are optional. CdServiceControl does not check for uppercase or lowercase letters.

pbRqData
cbRqData

are reserved for future functions that require additional information.

pbBuffer
cbBuffer

are reserved for future functions that return data.

psDataRet

are reserved for future functions that return data.

wFunction

specifies which control function the CD-ROM Service is to perform. A value of 1 specifies service deinstallation.

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33145
12	wFunction	2	
14	reserved	4	
18	pbRqData	4	
22	cbRqData	2	
24	pbDevSpec	4	
28	cbDevSpec	2	
30	pbBuffer	4	
34	cbBuffer	2	
36	psDataRet	4	
40	ssDataRet	2	2

This page intentionally left blank

CdVerifyPath (pbPathString, cbPathString): ercType

Description

CdVerifyPath verifies the existence of the specified CD-ROM path. Status code 15712 ("No such CD-ROM path") is returned if the path does not exist. No status code is returned if the path exists.

Using CdVerifyPath, an application can verify a file's existence and then open the file by calling CdOpen. (See CdOpen.)

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdVerifyPath (pbPathString, cbPathString): ercType

where

pbPathString
cbPathString

describe the path to be verified. Valid path names include CTOS specifications of the form

{node}[CDROMn]<Directory>FileName

or ISO heirarchical specifications of the form

{node}[CDROMn]\Directory\Subdirectory\FileName.

CdVerifyPath does not check for uppercase or lowercase letters when it matches file names.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33218
12	reserved	6	
18	pbPathString	4	
22	cbPathString	2	

CdVersion (pbDevSpec, cbDevSpec, pbVerInfo, cbVerInfo): ercType

Description

CdVersion returns the version level, hardware environment (workstation or SRP), and name of the CD-ROM Service that controls the specified device. In addition, this operation returns the number of CD-ROM drives currently installed.

CdVersion can return information on a CD-ROM Service that is installed remotely.

Status code 15702 ("Invalid specification") is returned if the device specification at *pbDevSpec* is invalid.

This operation is performed by the CD-ROM Service. (See "CD-ROM Service" in the *CTOS Programming Guide*.) The service must be installed for this operation to be used.

Procedural Interface

CdVersion (pbDevSpec, cbDevSpec, pbVerInfo, cbVerInfo): ercType

where

pbDevSpec
cbDevSpec

describe a character string of the form {Node}[DevName]. Square brackets are optional for the device name. CdVersion does not check for uppercase or lowercase letters.

pbVerInfo
cbVerInfo

describe the buffer to which the version information is to be returned. This information has the following format:

Offset	Field	Size (Bytes)	Description
0	wVersionLevel	2	The version level of the installed CD-ROM Service. The upper byte contains the revision number, and the lower byte contains the point release of the installed service. For example, if the CD-ROM Service is version 2.0, the upper byte is set to 2, while the lower byte is set to 0.
2	wHardwareEnv	1	The hardware environment of the installed CD-ROM Service. This value is 1 if the service is installed on a workstation. This value is 2 if the service is installed on a shared resource processor.
3	cDevices	1	The number of CD-ROM drives accessible to the CD-ROM service.
4	sbServiceName	13	The name under which the CD-ROM Service was installed. The service name is a character string in which the first byte is the size of the string. This name, in combination with the device number, forms the device name. For example, if the service name is CDROM, and the device's number is 1, the device's name would be [CDROM1].

Request Block

Offset	Field	Size (Bytes)	Contents
0	bCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33210
12	reserved	6	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pbVerInfo	4	
28	cbVerInfo	2	

This page intentionally left blank

CfaFfVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

CfaFfVersion returns a structure that identifies the version of the installed Cluster File Access (CFA) File Filter service.

Procedural Interface

CfaFfVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe a memory area to which the structure containing the version is written. The format of the structure is as follows:

Offset	Field	Size (Bytes)
0	version	2

pcbRet

describes a word into which the actual length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8226
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

CfaServerVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

CfaServerVersion returns a structure that identifies the version of the Cluster File Access (CFA) service installed at the server.

Procedural Interface

CfaServerVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe a memory area to which the structure containing the version is written. The format of the structure is as follows:

Offset	Field	Size (Bytes)
0	version	2

pcbRet

describes a word into which the actual length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8224
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

CfaWaVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

CfaWaVersion returns a structure that identifies the version of the installed Cluster File Access (CFA) Workstation Agent service and the configuration file name used to install the service. If a configuration file name was not entered during installation, the default name string [d0]<Sys>CfaConfig.sys is used to install the service, and this is the value returned.

Procedural Interface

CfaWaVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe a memory area to which the structure containing the version and configuration file name string is written. The format of the structure is as follows:

Offset	Field	Size (Bytes)
0	version	2
2	sbConfigFile	varies

The first byte of *sbConfigFile* is the length of the configuration file name string. The remaining bytes contain the string.

pcbRet

describes a word into which the actual length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8225
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

Chain (pbFileSpec, cbFileSpec, pbPassword, cbPassword, priority, ercTermination, fDebug): ercType

Description

Chain replaces a current application program with a specified run file. Chain returns control to the calling process only if an error condition is detected.

The process priority specified by Chain may be modified by SetDeltaPriority before the process is scheduled for execution. (See SetDeltaPriority.)

Chain performs the following functions:

1. Verifies that the specified run file exists, that it can be opened for read using the password provided, that it contains a valid arrangement of segments, and that it fits in the application partition.

2. Places the ercTermination (status code) in the Application System Control Block of the partition.

3. Disconnects interrupt handlers of the application partition and terminates all processes of the partition.

4. Terminates keyboard (in an application partition only), timer, and communications requests, and waits until all disk and printer input/output activity has ceased.

5. In an application partition only, resets the keyboard to character mode. Discards the content of the type-ahead buffer if the keyboard was in unencoded mode and/or the status code is nonzero.

6. In an application partition only, reenables the Action-Finish feature and discards the action code (if any).

7. In an application partition only, closes the Submit or recording file if the status code is nonzero.

8. Closes all files opened for the application partition except those marked long-lived by the OpenFileLL or SetFhLongevity operations. (See "File Management" in the *CTOS Operating System Concepts Manual* for more information.)

9. Releases for reuse all application partition memory that was allocated as short-lived.

10. In the partition, allocates a short-lived memory segment that is large enough to contain the specified run file.

11. Reads the run file into the application partition.

12. Relocates all intersegment references to accommodate the memory address at which the run file is loaded.

13. Creates a process to be scheduled at the specified priority. The initial values loaded into the segment registers (Code Segment (CS), Data Segment (DS), Stack Segment (SS), Extra Segment (ES)), the Stack Pointer (SP), and the Instruction Pointer (IP) are derived from information in the run-file header.

Chain has no effect on the allocation of long-lived memory. If the program requires virtual code segment management, the run file is left open to accommodate code swapping. The open run file handle is placed in the Application System Control Block of the application partition.

Procedural Interface

Chain (pbFileSpec, cbFileSpec, pbPassword, cbPassword, priority, ercTermination, fDebug): ercType

where

pbFileSpec
cbFileSpec

describe a character string of the following form: {Node}[VolName]<DirName>FileName.

pbPassword
cbPassword

describe either the volume, directory, or file password that authorizes access to the specified file.

priority

is the priority (0-254, with 0 the highest) at which to schedule the newly created process for execution.

ercTermination

is a 16-bit status code that is placed in the Application System Control Block of the application partition for examination by the run file. In an application partition only, a nonzero status code causes the content of the type-ahead buffer to be discarded and the Submit or recording file to be closed.

fDebug

if *fDebug* is TRUE, then the Debugger is entered automatically as soon as the run file is loaded into the application partition.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	28
12	priority	2	
14	ercTermination	2	
16	fDebug	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	

ChangeCommLineBaudRate (commLineHandle, iRxTx, baudRate): ercType

Description

ChangeCommLineBaudRate reinitializes the specified baud rate timer(s) initialized by InitCommLine. (See InitCommLine.) The channel is not reset.

ChangeCommLineBaudRate does not directly affect the serial communications controller chip if it is an 8274. However, errors may occur if this operation is performed while input/output is in progress.

On channels where the hardware does not support separate receive and transmit baud rates, *iRxTx* must be zero or status code 61 ("Comm line baud rates must match") is returned.

If the chip is an 8530, it will not recognize the new baud rate established by this operation until the time constant is reloaded after a countdown of the previous divisor. Baud rates are calculated assuming divide by 16 mode (in WR4). Operation in synchronous modes force the serial communications controller into divide by 1 mode, so when the serial controller is sourcing the clock, the baud rate should be programmed at 1/16th the desired rate.

Procedural Interface

ChangeCommLineBaudRate (commLineHandle, iRxTx, baudRate): ercType

where

commLineHandle

is a handle returned by InitCommLine.

iRxTx

is one of the following codes:

Code	Description
1	receive clocking only
2	transmit clocking only
0	both transmit and receive clocking

baudRate

is the baud rate, an integer (word), for example, 9600. Zero specifies external clocking.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4104
12	commLineHandle	2	
14	iRxTx	2	
16	baudRate	2	

This page intentionally left blank

ChangeFileLength (fh, lfaNewFileSize): ercType

Description

ChangeFileLength expands or contracts the file length to a new file length. In addition the end-of-file pointer in the File Header Block is set to the new length.

Procedural Interface

ChangeFileLength (fh, lfaNewFileSize): ercType

where

fh

is a file handle returned from an OpenFile operation. The file must be open in modify mode.

lfaNewFileSize

is the new file size in bytes. It must be a multiple of 512.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	13
12	fh	2	
14	lfaNewFileSize	4	

ChangeOpenMode (fh, pbPassword, cbPassword, modeNew): ercType

Description

ChangeOpenMode is used by a program to change the access mode of a file that is already open.

Status code 219 ("Access denied") is returned if the password supplied does not grant access to the file in the new mode. Status code 220 ("File in use") is returned if the new mode is mode modify and other users have the file open in mode read.

Procedural Interface

ChangeOpenMode (fh, pbPassword, cbPassword, modeNew): ercType

where

fh

is a file handle for a currently open file.

pbPassword
cbPassword

describe a password that gives access to the file in the new mode.

modeNew

is the new file open mode, which is read, modify, or peek. The mode is indicated by 16-bit values representing the ASCII constants "mr" (mode read), "mm" (mode modify), or "mp" (mode peek). In these ASCII constants, the first character (m) is the high-order byte and the second character is the low-order byte.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	215
12	fh	2	
14	modeNew	2	
16	reserved	2	
18	pbPassword	4	
22	cbPassword	2	

ChangePriority (priority): ercType

Description

The ChangePriority primitive changes the priority of the calling process. The priorities are 0-255, with 0 the highest. Priorities and their normal (and recommended) uses are as follows:

Priority	Use
0-9	operating system
10-64	system services
65-254	application programs
255	operating system background process only

If the priority specified by ChangePriority is 65-254, it may be modified by SetDeltaPriority before the process is scheduled for execution. (See SetDeltaPriority.)

If ChangePriority assigns a priority below 65 to an application program, the program may affect the normal operation of the operating system. For example, if the program is running under a partition-managing program such as the Context Manager, the Context Manager would not be able to modify the priority using SetDeltaPriority. Thus, the priority could conflict with priorities assigned to other partitions being managed.

Procedural Interface

ChangePriority (priority): ercType

where

priority is the new priority.

Request Block

ChangePriority is a Kernel primitive.

This page intentionally left blank

ChangeProcessPriority (pid, priority): ercType

Description

The ChangeProcessPriority primitive changes the priority of the process specified by the process ID. This operation is similar to ChangePriority.

The priority specified by ChangeProcessPriority may be modified by SetDeltaPriority before the process is scheduled for execution. (See SetDeltaPriority.)

Procedural Interface

ChangeProcessPriority (pid, priority): ercType

where

pid

is the process ID (returned by NewProcess) of the process whose priority is to be altered.

priority

specifies the new value (0-254) of the priority.

Request Block

ChangeProcessPriority is a Kernel primitive.

This page intentionally left blank

Change user number requests have no procedural interface.

NOTE: This page presents the standard format for all change user number requests.

Description

Change user number requests are required only by multipartition (MP) operating sytems. (Appendix A in the *CTOS Operating System Concepts Manual* indicates which operating systems are MP operating systems.) In MP systems, the primary partition always is user number 1. When a program converts to a system service by calling ConvertToSys, a new system partition is created out of part of the primary partition. To maintain user number 1 as the primary partition user number, a new user number is assigned to the system partition created. As a result, change user number requests need to be issued to all existing system services to notify them of the new user number.

(See Appendix D for a list of the operating system change user number requests and request codes that are issued.)

Procedural Interface

Change user number requests have no procedural interface.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	newUserNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	*
12	oldUserNum	2	

*Request code of change user number request

Check (exchange, ppMsgRet): ercType

Description

The Check primitive checks whether messages are queued at the specified exchange. If messages are queued, then the message that was queued first is removed from the queue and its memory address is returned to the calling process. If no messages are queued, then status code 14 ("No message available") is returned.

The Check primitive, unlike the Wait primitive, never causes the calling process to be placed into the waiting state.

Procedural Interface

Check (exchange, ppMsgRet): ercType

where

exchange

is the identification of the exchange to check.

ppMsgRet

is the memory address of a 4-byte field where the memory address of the message that was queued first at the exchange, if any, is returned.

Request Block

Check is a Kernel primitive.

This page intentionally left blank

CheckContextStack: ercType

Description

CheckContextStack is used by an asynchronous system service to validate the context stack. All the asynchronous request procedures (BuildAsyncRequest, BuildAsyncRequestDirect, AsyncRequest, and AsyncRequestDirect) ensure that the context stack has not overflowed. If, however, the asynchronous system service you have written calls a procedure other than these, it should also call CheckContextStack to ensure that the stack has not overflowed.

Procedural Interface

CheckContextStack: ercType

Request Block

CheckContextStack is an object module procedure in Async.lib.

This page intentionally left blank

CheckErc (erc)

Description

CheckErc is used to check status codes. If CheckErc is called with a non-zero status code, then FatalError is called with that value.

This operation is used, for example, to replace the program segment

```
erc = CloseFile(fh);
    if (erc !=0) FatalError(erc);
```

with

```
CheckErc(CloseFile(fh));
```

Procedural Interface

CheckErc (erc)

where

erc

is a status code.

Request Block

CheckErc is an object module procedure.

This page intentionally left blank

CheckForOperatorRestartC (pBs, pfRestartRet): ercType

Description

CheckForOperatorRestartC is used to check for an operator signal to restart a printer.

CheckForOperatorRestartC returns TRUE if, since the byte stream was opened or the last time characters were written (e.g., FlushBufferC) or CheckForOperatorRestartC was called, either a break has been received or (if XON/XOFF line control is in effect) an XON has been received while the transmit queue was empty.

These occurrences are indications that an operator has manually signaled that output to a serial printer should resume, either by pressing the printer's break switch or by opening and then closing the printer cover after output has ceased.

The Spooler, for example, can call this operation periodically to restart output in response to manual intervention.

(See "Communications Programming" in the *CTOS Operating System Concepts Manual.)*

Procedural Interface

CheckForOperatorRestartC (pBs, pfRestartRet): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

pfRestartRet

is the memory address of a flag that indicates whether the operator has signaled that output should resume.

Request Block

CheckForOperatorRestartC is an object module procedure.

CheckpointBs (pBswa): ercType

Description

CheckpointBs checkpoints the open output byte stream identified by the memory address of the Byte Stream Work Area. CheckpointBs writes any partially full buffers and waits for all write operations to complete successfully before returning. The byte stream remains open for subsequent output.

Procedural Interface

CheckpointBs (pBswa): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

Request Block

CheckpointBs is an object module procedure.

This page intentionally left blank

CheckpointBsAsyncC (pBs, fIsPartOfClose, idExch,pMsg, rqCode): ercType

Description

CheckpointBsAsyncC is the asynchronous (nonblocking) form of CheckPointBsC. (See CheckPointBsC.) It can be used to perform the CheckPointBsC function when the caller does not want his process to wait.

Procedural Interface

CheckpointBsAsyncC (pBs, fIsPartOfClose, idExch, pMsg, rqCode): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

fIsPartOfClose

is a flag. If *fIsPartOfClose* is TRUE and the byte stream is a serial printer byte stream, carriage return/linefeed is appended.

idExch
pMsg
rqCode

are the same parameters found in FlushBufferAsyncC. (See FlushBufferAsyncC.) If their values are zero, CheckpointBsAsyncC is the same operation as CheckPointBsC.

Request Block

CheckpointBsAsyncC is an object module procedure.

This page intentionally left blank

CheckPointBsC (pBs, fIsPartOfClose): ercType

Description

CheckPointBsC blocks (waits) until all characters previously written to the byte stream have been physically output from the serial port.

In addition, CheckPointBsC can be used to append carriage return/line feed to the output. This is useful in order to force a buffered printer to print the last line of output.

(See "Communications Programming" in the *CTOS Operating System Concepts Manual.)*

Procedural Interface

CheckPointBsC (pBs, fIsPartOfClose): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

fIsPartOfClose

is a flag. If *fIsPartOfClose* is TRUE, and the byte stream is a serial printer byte stream, and image mode is not in effect, carriage return/line feed is appended. The name of this parameter comes from the fact that CheckPointBsC is called as part of the CloseByteStream operation with this argument set to TRUE. Note that serial printer byte streams include any byte stream opened using a "[ptr]" device specification, regardless of configuration, as well as any byte stream configured as a serial printer byte stream (using a serial printer configuration file or the equivalent AcquireByteStreamC configuration structure).

Request Block

CheckPointBsC is an object module procedure.

This page intentionally left blank

CheckpointRsFile (pRswa): ercType

Description

CheckpointRsFile checkpoints the open output RSAM file identified by the memory address of the Record Sequential Work Area. CheckpointRsFile writes any partially full buffers and waits for all write operations to complete before returning. The RSAM file remains open for subsequent output.

Procedural Interface

CheckpointRsFile (pRswa): ercType

where

pRswa

is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

Request Block

CheckpointRsFile is an object module procedure.

This page intentionally left blank

CheckpointSysIn: ercType

Description

CheckpointSysIn writes the content of the current, partially filled, output buffer to the recording file if the System Input Process is in recording mode. If the System Input Process is in normal or playback mode, no action occurs.

Procedural Interface

CheckpointSysIn: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	68

This page intentionally left blank

CheckReadAsync (pRq, psDataRet): ercType

Description

CheckReadAsync synchronizes a process with an asynchronous read. For example, after calling ReadAsync to initiate a Read, the requesting process continues execution. When the process wants to synchronize with the asynchronous read (that is, wait for its completion), the process calls CheckReadAsync. CheckReadAsync waits for input completion, checks the status code, and obtains the byte count of the data read.

Status code 248 ("Wrong *pRq* argument") is returned if the *pRq* argument does not match the argument of the preceding ReadAsync.

Procedural Interface

CheckReadAsync (pRq, psDataRet): ercType

where

pRq

is the same memory address as given in the *pRq* argument of the ReadAsync operation.

psDataRet

is the memory address of the word where the count of bytes successfully read is to be returned.

Request Block

ReadAsync and CheckReadAsync are procedural interfaces to the Read operation. (See Read.)

This page intentionally left blank

CheckWriteAsync (pRq, psDataRet): ercType

Description

CheckWriteAsync synchronizes a process with an asychronized write. For example, after calling WriteAsync to initiate a write, the requesting process continues execution. When the process wants to synchronize with the asynchronous write (that is, wait for its completion), the process calls CheckWriteAsync. CheckWriteAsync waits for output completion, checks the status code, and obtains the byte count of data written.

Status code 248 ("Wrong *pRq* argument") is returned if the *pRq* argument does not match the argument of the preceding WriteAsync.

Procedural Interface

CheckWriteAsync (pRq, psDataRet): ercType

where

pRq

is the same memory address as given in the *pRq* argument of the WriteAsync operation.

psDataRet

is the memory address of the word where the count of bytes successfully written is to be returned.

Request Block

WriteAsync and CheckWriteAsync are procedural interfaces to the Write operation. (See Write.)

This page intentionally left blank

CleanQueue(qH): ercType

Description

The CleanQueue operation allows an application program to clean out one of its queues in case of data corruption. The specified queue is reset to empty.

Procedural Interface

CleanQueue(qH): ercType

where

qH

is a word that contains the queue handle returned by the Queue Manager to the application program that installed the queue.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32866
12	qH	2	

This page intentionally left blank

ClearPath: ercType

Description

ClearPath clears the defaults established by the SetPath and SetPrefix operations. (For details, see the descriptions of SetPath and SetPrefix.)

Procedural Interface

ClearPath: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	2

This page intentionally left blank

CloseAllFiles: ercType

Description

CloseAllFiles closes all files that are currently open for the user, except those marked long-lived (opened with the OpenFileLL operation).

Procedural Interface

CloseAllFiles: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	19

This page intentionally left blank

CloseAllFilesLL: ercType

Description

CloseAllFilesLL closes all files that are currently open for the user, including those marked long-lived. The CloseAllFiles operation, on the other hand, closes all open files except those marked long-lived. (See CloseAllFiles.)

Procedural Interface

CloseAllFilesLL: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	62

This page intentionally left blank

CloseAltMsgFile (pMwa, sMwa): ercType

Description

CloseAltMsgFile closes an open message file on disk. CloseAltMsgFile is similar to CloseMsgFile except that, with alternate message files, any number of message files may be open at any time. Access to each message file is determned by the Message Work Area (MWA).

Procedural Interface

CloseAltMsgFile (pMwa, sMwa): ercType

where

pMwa

is the memory address of the MWA that was supplied to InitAltMsgFile.

sMwa

is the size (word) of the MWA. *sMwa* must be at least 60 bytes.

Request Block

CloseAltMsgFile is an object module procedure.

This page intentionally left blank

CloseByteStream (pBswa): ercType

Description

CloseByteStream closes the open byte stream identified by the memory address of the Byte Stream Work Area. If the byte stream was open for output, then CloseByteStream writes any partially full buffers and waits for all write operations to complete before returning. After calling CloseByteStream, the process can reuse the Byte Stream Work Area and buffer area. If an error occurs during a CloseByteStream operation, then the byte stream is closed and a status code is returned.

Procedural Interface

CloseByteStream (pBswa): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

Request Block

CloseByteStream is an object module procedure.

This page intentionally left blank

CloseDaFile (pDawa): ercType

Description

CloseDaFile closes the open DAM file identified by the memory address of the Direct Access Work Area. After calling CloseDaFile, the application program can reuse the Direct Access Work Area and the buffer area.

Procedural Interface

CloseDaFile (pDawa): ercType

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

Request Block

CloseDaFile is an object module procedure.

This page intentionally left blank

CloseErcFile: ercType

Description

CloseErcFile closes an error message file on disk that was previously opened using InitErcFile. CloseErcFile is similar to CloseMsgFile except that, with CloseErcFile, closing the error message file does not affect the application's normal message file.

Procedural Interface

CloseErcFile: ercType

Request Block

CloseErcFile is an object module procedure.

This page intentionally left blank

CloseFile (fh): ercType

Description

CloseFile closes an open file. Note that CloseFile does not alter the end-of-file pointer. (See SetFileStatus.)

NOTE: Close is a synonym for CloseFile and may be used in its place. If, however, Close is a keyword in the language in which a program is coded, using this word will result in duplicate external references.

Procedural Interface

CloseFile (fh): ercType

where

fh

is the file handle returned from an OpenFile operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	10
12	fh	2	

CloseMsgFile: ercType

Description

CloseMsgFile closes an open message file on disk.

Procedural Interface

CloseMsgFile: ercType

Request Block

CloseMsgFile is an object module procedure.

This page intentionally left blank

ClosePSSession (sh): ercType

Description

ClosePSSession is a synonym for PSCloseSession. See the description of PSCloseSession for details.

This page intentionally left blank

CloseRsFile (pRswa): ercType

Description

CloseRsFile closes the open RSAM file identified by the memory address of the Record Sequential Work Area. If the RSAM file was open for output, CloseRsFile writes any partially full buffers and waits for all write operations to complete before returning. After calling CloseRsFile, the Record Sequential Work Area and buffer area can be reused. If an error occurs during a CloseRsFile operation, the RSAM file is closed and the pertinent status code is returned.

Procedural Interface

CloseRsFile (pRswa): ercType

where

pRswa

is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

Request Block

CloseRsFile is an object module procedure.

This page intentionally left blank

CloseRtClock (pRqTime): ercType

Description

CloseRtClock terminates the use of the specified Timer Request Block (TRB).

(See Chapter 4, "System Structures," for the format of a TRB.)

Procedural Interface

CloseRtClock (pRqTime): ercType

where

pRqTime

is the memory address of a TRB that is currently open.

Request Block

sRqTime is always 12.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	50
12	reserved	6	
18	pRqTime	4	
22	sRqTime	2	12

CloseServerMsgFile: ercType

Description

CloseServerMsgFile closes a message previously opened by a call to OpenServerMsgFile. This operation also deallocates memory allocated by OpenServerMsgFile.

(See the related operation, OpenServerMsgFile.)

Procedural Interface

CloseServerMsgFile: ercType

Request Block

CloseServerMsgFile is an object module procedure.

This page intentionally left blank

CloseTerminal has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *This operation works on a shared resource processor hardware only. Do not use it on workstation hardware.*

Description

CloseTerminal is sent by a client to indicate that it is finished with the port on either a Cluster or Terminal Processor. It causes any pending ReadTerminal operation to be returned with an error.

If the line is connected to a data set and NoHangUpOnClose has not been selected, CloseTerminal places it back on-hook in anticipation of a new call. If NoHangUpOnClose is selected, CloseTerminal does not deassert Data Terminal Ready (DTR).

Procedural Interface

CloseTerminal has no procedural interface. You must make a request block and issue the request using one of the request operations listed above.

where

bDestCpu

is the slot number of the Cluster or Terminal Processor to which the asynchronous terminal is connected.

bSourceCpu

is the slot number of the client.

bPort

is the port number of a terminal attached to an RS-232-C port.

bWindow

must be 0.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	ExchResp	2	
8	ercRet	2	
10	rqCode	2	12307
12	bDestCPU	1	
13	bSourceCPU	1	
14	bPort	1	
15	bWindow	1	

CloseVidFilter: ercType

Description

CloseVidFilter clears the flag set in the Application System Control Block by a previous call to the OpenVidFilter operation and closes the video byte stream. (See the description of OpenVidFilter for details.)

OpenVidFilter and CloseVidFilter are used to suppress the video output in a program to which the caller chains. The Batch program, for example, uses these routines to suppress video output in programs it runs.

Procedural Interface

CloseVidFilter: ercType

Request Block

CloseVidFilter is an object module procedure.

This page intentionally left blank

CloseSysCmds: ercType

Description

CloseSysCmds closes the Executive command file that was previously opened by the InitSysCmds operation. (See InitSysCmds.)

Procedural Interface

CloseSysCmds: ercType

Request Block

CloseSysCmds is an object module procedure.

This page intentionally left blank

CompactDateTime (pExpDateTime, pDateTimeRet): ercType

Description

CompactDateTime converts the date and time from the Expanded Date/Time format to the System Date/Time format. Each field of the Expanded Date/Time format is verified. If the day of week does not agree with the other fields, status code 2702 ("Day and date disagree") is returned.

(For details on the expanded format, see "Expanded Date/Time Format" in Chapter 4, "System Structures." The System Date/Time format is described in "Utility Operations" in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

CompactDateTime (pExpDateTime, pDateTimeRet): ercType

where

pExpDateTime

is the memory address of an 8-byte block in the Expanded Date/Time format.

pDateTimeRet

is the memory address to which the 32-bit date/time in the System Date/Time format is to be returned.

Request Block

CompactDateTime is an object module procedure.

This page intentionally left blank

ConfigCloseFile: ercType

Description

ConfigCloseFile closes a textual configuration file that was opened previously by the ConfigOpenFile operation. (See ConfigOpenFile.)

If the application needs to open a new configuration file, it must first use ConfigCloseFile to close the configuration file currently in use.

Status code 1100 ("File not open") is returned if ConfigOpenFile was not previously called.

For details on configuration file operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

ConfigCloseFile: ercType

Request Block

ConfigCloseFile is an object module **procedure.**

This page intentionally left blank

ConfigGetNextToken (psbToken, cbTokenMax, bStart, bEnd, fNewLine): ercType

Description

ConfigGetNextToken returns the next field in a configuration file that begins and ends with the specified delimiter. Unlike LookUpField, ConfigGetNextToken can return fields that are delimited by characters other than colons (for example, slash marks).

Note that the configuration file must have been previously opened by the ConfigOpenFile operation. (See ConfigOpenFile.)

ConfigGetNextToken begins its search at the position last reached by a previous ConfigGetRestOfLine or ConfigSetPosition operation. After the field has been located, ConfigGetRestOfLine can be used to obtain the remaining contents of the line. (See ConfigGetRestOfLine and ConfigSetPosition.)

ConfigGetNextToken does not return the delimiters with the field. If ConfigGetNextToken encounters a pair of quotes inside the delimiters, it does not include the quotes with the returned text. For example, suppose ConfigGetNextToken encounters the following field in a configuration file:

/user name "for example, John"/

If the application specifies a slash as the value of the starting and ending delimiter parameters, ConfigGetNextToken returns the following string:

user name for example, John

In certain cases, the application may need need to obtain the double quotes and the quoted characters within the field. ConfigGetNextToken returns these characters if they are enclosed by single quotes ('). Suppose ConfigGetNextToken encounters the following field:

/user name '"for example, John"'/

In this case, ConfigGetNextToken returns the following string:

user name "for example, John"

When *fNewLine* is set to TRUE, ConfigGetNextToken skips to the next line if it encounters characters between the carriage return and the first delimiter of the field. Consider the following example:

/Field1Name/ 12 13

/Field2Name/ 14

/Field3Name/ 15

In this example, suppose *fNewLine* is set to TRUE and the slash (/) is specified as the value of the delimiters. The first call to ConfigGetNextToken returns the string, Field1Name. When ConfigGetNextToken is called a second time with the same parameters, it returns Field3Name. The string Field2Name is not returned because it is preceeded by a tab character.

Status code 1 ("End of file") is returned if the file position reached is at the end of the file. Status code 1100 ("Configuration file not open") is returned if the configuration file is not open.

For details on configuration file operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual*.

Procedural Interface

ConfigGetNextToken (psbToken, cbTokenMax, bStart, bEnd, fNewLine): ercType

where

psbToken
cbTokenMax

describe a memory area to which the string separated by the given delimiters is returned. The first byte contains the length of the string.

bStart

is the starting delimiter (for example, a colon or slash). If *bStart* is a space (' '), ConfigGetNextToken skips all space, tab, and null characters until it reaches text.

bEnd

is the ending delimiter (for example, a colon). If *bEnd* is a space, ConfigGetNextToken returns all text up to the next space, tab, or null character.

fNewLine

is a flag that, if TRUE, causes ConfigGetNextToken to retrieve the next string, preceded by the specified starting delimiter, that follows a new line character. To locate standard CTOS configuration file entries, the caller should set this value to TRUE.

Request Block

ConfigGetNextToken is an object module procedure.

This page intentionally left blank

ConfigGetPosition (pLfaRet): ercType

Description

ConfigGetPosition returns the position within the configuration file where the next ConfigGetNextToken or ConfigGetRestOfLine operation will read text. Note that the configuration file must have been previously opened by the ConfigOpenFile operation. (See ConfigOpenFile.)

The returned position is either the beginning of the file, or the point last reached by the most recent call to ConfigGetNextToken or ConfigGetRestOfLine. To return to this position at a later time, an application may call ConfigSetPosition with the recorded position provided by ConfigGetPosition. (See ConfigSetPosition.)

For details on configuration file operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual*.

Procedural Interface

ConfigGetPosition (pLfaRet): ercType

where

pLfaRet

is the memory address of a double word to which the current file position is returned.

Request Block

ConfigGetPosition is an object module procedure.

This page intentionally left blank

ConfigGetRestOfLine (psbToken, cbTokenMax, fSkipWhite): ercType

Description

ConfigGetRestOfLine returns the contents of a configuration file, from the file position last reached by a previous ConfigGetNextToken operation to the end of the current line. The configuration file must have been previously opened by the ConfigOpenFile operation. (See ConfigOpenFile.)

When *fSkipWhite* is TRUE, ConfigGetRestOfLine does not include any leading space, tab, or null characters with the returned text.

If ConfigGetRestOfLine encounters a pair of quotes (either ' or "), it does not include the quotes with the returned text. The following is an example of a line in a configuration file:

```
:username: 'John' 'Sam'
```

In this example, suppose the most recent ConfigGetNextToken operation returned the field, username. When ConfigGetRestOfLine is called with *fSkipWhite* set to TRUE, it returns the following string:

```
John Sam
```

Status code 1 ("End of file") is returned if the file position last reached is at the end of the file.

For details on configuration file operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual*.

Procedural Interface

ConfigGetRestOfLine (psbToken, cbTokenMax, fSkipWhite): ercType

where

psbToken
cbTokenMax

describe a memory area where the string read from the configuration file is to be returned. The first byte of the data returned contains the length of the string.

fSkipWhite

is a flag which, when TRUE, indicates that any leading space, tab, or null characters should not be returned with the string.

Request Block

ConfigGetRestOfLine is an object module procedure.

ConfigOpenFile (pbSpec, cbSpec, pbBuffer, cbBuffer): ercType

Description

ConfigOpenFile opens a standard or nonstandard configuration file, thus allowing it to be accessed by the ConfigGetNextToken, ConfigGetRestOfLine, ConfigGetPosition, and ConfigSetPosition operations. Only one configuration file may be opened at a time.

After the application opens and accesses a configuration file, it can close the file by calling ConfigCloseFile. (See ConfigCloseFile.)

Status code 1101 ("File already open") is returned if a configuration file has already been opened.

For details on configuration file operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual*.

Procedural Interface

ConfigOpenFile (pbSpec, cbSpec, pbBuffer, cbBuffer): ercType

where

pbSpec
cbSpec

describe the file specification for the configuration file to be opened.

pbBuffer
cbBuffer

describe a word-aligned I/O buffer. *cbBuffer* must be at least 512 bytes in size.

Request Block

ConfigOpenFile is an object module procedure.

ConfigSetPosition (lfaNew): ercType

Description

ConfigSetPosition sets the position, within the configuration file, where the next ConfigGetNextToken or ConfigGetRestOfLine operation will read text. The configuration file must have been previously opened by the ConfigOpenFile operation. (See ConfigOpenFile.)

The caller-specified position must be either zero or a value recorded by a previous ConfigGetPosition operation.

Status code 1 ("End of file") is returned if the file position last reached is at the end of the file.

For details on configuration file operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

ConfigSetPosition (lfaNew): ercType

where

lfaNew

is the new double-word file position from which data is to be read by a subsequent file operation.

Request Block

ConfigSetPosition is an object module procedure.

This page intentionally left blank

ConfigureSpooler (channel, pbPrinterName, cbPrinterName, pbQueueName, cbQueueName, pbConfigureFile, cbConfigureFile, priority, fBanner): ercType

Description

ConfigureSpooler sets or changes the spooler's configuration. A printer can be added or deleted from a spooler. To add a printer, the printing queue associated with the printer must be defined to the Queue Manager and the printer channel must be defined to the spooler during the spooler's installation.

Procedural Interface

ConfigureSpooler (channel, pbPrinterName, cbPrinterName, pbQueueName, cbQueueName, pbConfigureFile, cbConfigureFile, priority, fBanner): ercType

where

channel

is a single-character code that specifies the printer channel to which the printer is connected. Examples are

Code	Description
0	parallel printer channel
A	channel A
B	channel B

pbPrinterName
cbPrinterName

describe the name of the printer to be added. A "0" means the printer connected to the channel is deleted.

pbQueueName
cbQueueName

describe the name of the scheduling queue associated with the printer. The name must match a queue name defined for the system.

pbConfigureFile
cbConfigureFile

describe the file specification of the printer configuration file.

priority

is the priority (10-24, with 10 the highest) of the spooler's control process for the printer. A priority lower than 128 (the default priority of the user program) ensures that the spooler does not impact the user program.

fBanner

is a flag that indicates whether a banner page is to be printed at the beginning of each file.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	188
12	channel	1	
13	priority	1	
14	fBanner	1	
15	reserved	3	
18	pbPrinterName	4	
22	cbPrinterName	2	
24	pbQueueName	4	
28	cbQueueName	2	
30	pbConfigureFile	4	
34	cbConfigureFile	2	

This page intentionally left blank

ControlInterrupt (tyDev, bOp): ercType

Description

The ControlInterrupt Kernel primitive enables, masks, or ends an interrupt for a specified device in a device independent manner. The advantage of using this operation is that it eliminates the need to specify port addresses or other details of a particular programmable interrupt controller. As a result, a program using this operation can be ported to other systems with different hardware requirements.

Procedural Interface

ControlInterrupt(tyDev, bOp): ercType

where

tyDev

is the type of the device (word). Each type is a physical line from a device to an interrupt controller. *tyDev* can be any of the following values:

Value	Device
1	reserved for use by the operating system
2	reserved for use by the operating system
3	Programmable Interval Timer
4	Realtime clock
5	parallel printer
6	keyboard
7	first RS-422-C cluster line on workstations or on a shared resource processor CP or GP
8	second RS-422-C cluster line on a shared resource processor GP or CP
9	floppy diskette
10	X-Bus SCSI module for workstations; ST-506 disk for workstations and for a shared resource processor FP
11	internal SCSI adapter for Series i workstations; SCSI adaptor 0 for a shared resource processor GP+SI
12	SCSI adaptor 1 for a shared resource processor GP+SI
13	82380 DMA Terminal Count (TC) for a shared resource processor GP

Value	Device
14	doorbell
15	communication device channel A and B
16	communication device channel C and D
17	communication device channel E and F
18	communication device channel G and H
19	coprocessor
20	redirect interrupt channel dedicated to communication between cascaded interrupt controllers (see note below)
22	shared resource processor bus DMA
23	error correction code (ECC) error
24	X-Bus interrupt 0
25	X-Bus interrupt 1
26	X-Bus floppy diskette on SuperGen Series 5000
27	X-Bus interrupt 3
28	X-Bus interrupt 4
29	HINT R87 X-Bus interrupt handler (OEM only)
30	keyboard on SuperGen Series 5000
31	X-Bus timeout on SuperGen Series 5000
32	video/voice cartridge on SuperGen Series 5000

bOp

is a byte value indicating the interrupt controlling operation. *bOp* is one of the following values:

Value	Operation
0	enable interrupt
1	mask interrupt
2	end-of-interrupt (EOI). With this value, *tyDev* is ignored.

Request Block

ControlInterrupt is a Kernel primitive.

ConvertToSys: ercType

Caution: *This operation should not be used by real mode programs executing on a protected mode operating system.*

Description

ConvertToSys converts all processes, short-lived memory, and exchanges in an application partition to system service processes, system memory, and system exchanges, respectively, in a system partition.

This operation is legal if *only one* application partition exists in memory. If more than one application partition exists, status code 810 ("Invalid Request") is returned.

NOTE: Long-lived memory and unallocated memory are not part of the system partition.

Procedural Interface

ConvertToSys: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	98

CopyFile (fh, pbToFileSpec, cbToFilespec, pbPassword, cbPassword, pBuffer, sBuffer, fOverwriteOk): ercType

Description

CopyFile creates a copy of the current contents of an open file. CopyFile is most commonly used to create a -Old file before modifying a file.

To copy the file as quickly as possible, CopyFile attempts to allocate more memory than the buffer passed to it. If CopyFile can allocate more memory, it does so and then deallocates it before returning to the caller.

To conserve memory used by the program, CopyFile does not use the Sequential Access Method, which would require linking byte streams with the program. Instead, CopyFile can be used by any of the structured file access methods. (For details, see "Structured File Access Methods" in the *CTOS Operating System Concepts Manual.*) When you use a structured file access method in your program that buffers data to be written to the file, however, be sure your program flushes the buffer before calling CopyFile.

Procedural Interface

CopyFile (fh, pbToFileSpec, cbToFilespec, pbPassword, cbPassword, pBuffer, sBuffer, fOverwriteOk): ercType

where

fh

is the file handle (word) of the file to be copied.

pbToFileSpec
cbToFileSpec

describe a character string of the form {Node}[VolName]<DirName>Filename to be used as the file specification of the new file.

pbPassword
cbPassword

describe the volume or directory password that authorizes the creation of the new file or access to an existing file to be overwritten. The password is not a password to be assigned to the file being created. The file password of the original file is assigneed to the file copy.

pBuffer
sBuffer

describe a buffer to be used while copying. *sBuffer* must be a multiple of 512.

fOverOk

is a flag (byte). TRUE causes the file to be overwritten with the contents of the open file. FALSE will return an error indicating that the file already exists.

Request Block

CopyFile is an object module procedure.

CParams: WORD

Description

CParams returns the number of parameters in the Variable Length Parameter Block (VLPB). The most common use of the VLPB is to store parameters from an Executive command.

Note that the Executive passes the name of the command in subparameter 0 of parameter 0.

Procedural Interface

CParams: WORD

Request Block

CParams is an object module procedure.

This page intentionally left blank

Crash (ercTermination)

Description

On workstations, Crash causes the operating system operation to terminate, and the following events to happen:

1. A crash dump is written to a file, such as

   the [Sys]<Sys>CrashDump.Sys file of a workstation with a local file system

   or the [Sys]<Sys>WS>CrashDump.sys file of the server to which a diskless workstation is attached

2. The operating system is reloaded.

The SignOn form displays the cause of the crash when the operating system is restarted. Crash never returns.

On a shared resource processor, the results of the Crash operation are different. For details, see the "Crash Analysis" section in the *CTOS Status Ccdes Reference Manual.*

Application programs should not call Crash except in cases of catastrophic events. To terminate an application program, use the FatalError operation instead. FatalError terminates a program while allowing other programs to continue executing.

Procedural Interface

Crash (ercTermination)

where

ercTermination

is a 16-bit status code indicating the cause of the crash.

Request Block

Crash is a system-common procedure.

CreateAlias(pSource, pSnRet): ercType

Description

CreateAlias returns an alias selector for the specified source memory address, *pSource*. The alias selector combined with a zero offset references the same linear address as *pSource*.

Programs that execute in both real and protected modes should use this operation rather than using pointer arithmetic to create short pointers, that is, memory addresses having zero offsets.

In protected mode CreateAlias allocates a descriptor table entry.

Procedural Interface

CreateAlias(pSource, pSnRet): ercType

where

pSource

is the memory address to which an alias is to be created. *pSource* must be paragraph aligned in real mode.

pSnRet

is the memory address of a word to which the alias selector is to be returned.

Request Block

sSnMax is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	284
12	reserved	2	
14	pSource	2	
18	pSnRet	4	
22	sSnMax	2	2

CreateBigPartition (pbPartitionName, cbPartitionName, qParagraph, bParType, pUserNumRet): ercType

Description

CreateBigPartition is the same as CreatePartition except that CreateBigPartition allows you to create a new application partition that is larger than 1M byte. (See CreatePartition.)

Procedural Interface

CreateBigPartition (pbPartitionName, cbPartitionName, qParagraph, bParType, pUserNumRet): ercType

where

pbPartitionName
cbPartitionName

describe the partition name (up to 12 characters).

qParagraph

is a quad that indicates the number of paragraphs of memory to be allocated to the application partition. If *qParagraph* is 0, then the partition is allocated all of the available memory.

If *bParType* is 6, *qParagraph* is the maximum number of paragraphs available to the partition and can be overridden by a smaller value.

bParType

is a byte that specifies an application's partition memory requirements including overhead for buffers, keyboard, or screen, for example.

bParType is compatible with the 0 and 0FFh values of the *fRunBatch* parameter of real mode operating systems. 0FFh indicates that a Batch Control Block of 1.3K bytes is allocated in addition to the memory for the partition itself. 0 indicates no Batch Control Block is allocated.

In addition, *bParType* has the values of 2 or 6, which are used only by partition-managing programs, such as the Context Manager. 2 indicates the creation of a partition, including virtual screen and keyboard structures with no resizing allowed upon chain; 6 allows, in addition, resizing upon chain.

pUserNumRet

is the memory address of a word where the user number of the created partition is returned.

Request Block

sUserNumRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nResPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	331
12	qParagraphLow	2	
14	bParType	1	
15	reserved	1	
16	qParagraphHigh	2	
18	pbPartitionName	4	
22	cbPartitionName	2	
24	pUserNumRet	4	
28	sUserNumRet	2	2

This page intentionally left blank

CreateContext(stackSize, userNum): ercType

Description

CreateContext is used by an asynchronous system service to allocate stack space for a context from the heap. To conserve heap memory, stack size for a context is limited to approximately 300 bytes. If no memory is available from the heap, status code 4533 ("Heap memory not available") is returned.

When control returns from CreateContext with a status code of 0 ("ercOK"), the stack pointer is changed. The caller, therefore, must not use any local variables upon the return because they are no longer addressable.

Procedural Interface

CreateContext(stackSize, userNum): ercType

where

stackSize

is the number of bytes to allocate for the stack of the context to be created.

userNum

is the user number associated with the context. If the context is associated with a Timer Request block (TRB) rather than a client request, *userNum* should be 0.

Request Block

CreateContext is an object module procedure in Async.lib.

This page intentionally left blank

CreateDir (pbDirSpec, cbDirSpec, pbVolPassword, cbVolPassword, pbDirPassword, cbDirPassword, cSectors, defaultFileProtectionLevel): ercType

Description

CreateDir creates a directory of the specified name on the specified volume. The volume name can be defaulted to that specified in a previous SetPath operation.

Status code 240 ("Directory already exists") is returned if a directory of the specified name already exists.

Procedural Interface

CreateDir(pbDirSpec, cbDirSpec, pbVolPassword, cbVolPassword, pbDirPassword, cbDirPassword, cSectors, defaultFileProtectionLevel): ercType

where

pbDirSpec
cbDirSpec

describe a character string of the form {Node}[VolName]DirName.

pbVolPassword
cbVolPassword

describe the volume password. If the CreatedirectoryProtection field in the system configuration file Config.sys is set to 'no', CreateDir does not require a volume password. In such a case, the caller may set these parameter values to 0.

pbDirPassword
cbDirPassword

describe the directory password to be assigned to this directory.

cSectors

is the size of the directory in 512-byte sectors. The number of directory entries per sector depends on the length of the file names of the files created in the directory. You can compute an approximate value for *cSectors* by adding 24 to the maximum number of files you expect to create in the directory and dividing the result by 25. For example, if you expect to create 750 files,

(750+24)/25 = 30 sectors

defaultFileProtectionLevel

is the default file protection level to be assigned to files in this directory. (Protection levels are listed in "File Management" in the *CTOS Operating System Concepts Manual.*)

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	17
12	reserved	2	
14	cSectors	2	
16	defaultFileProtectionLevel	2	
18	pbDirSpec	4	
22	cbDirSpec	2	
24	pbVolPassword	4	
28	cbVolPassword	2	
30	pbDirPassword	4	
34	cbDirPassword	2	

CreateExecScreen (sVidMap, cColsScreen, cLinesScreen): ercType

Description

CreateExecScreen sets up the video display in the same style as the Executive screen. Three video frames (Command, Status, and Event) are initialized and the Event Frame double-bar is drawn.

Procedural Interface

CreateExecScreen (sVidMap, cColsScreen, cLinesScreen): ercType

where

sVidMap

is the size (word) of the video map. This value is returned from a previous call to the ResetVideo operation. ResetVideo must be called before calling CreateExecScreen.

cColsScreen

is the maximum number (word) of video columns. This value is returned by the QueryVideo or QueryVidHdw operation.

cLinesScreen

is the maximum number (word) of video lines. This value is returned by the QueryVideo or QueryVidHdw operation.

Request Block

CreateExecScreen is an object module procedure.

This page intentionally left blank

CreateFile (pbFileSpec, cbFileSpec, pbPassword, cbPassword, lfaFileSize): ercType

Description

CreateFile creates a file of the specified name and size in the specified directory on the specified volume.

CreateFile creates a file; it does not open it. Use OpenFile to open a file after it is created.

Status code 224 ("File already exists") is returned if a file of the specified name already exists.

By default, the file created inherits the protection level and password assigned to the directory in which it was created. (See the decription of CreateDir. For details on protection levels, see "File Management" in the *CTOS Operating System Concepts Manual*.)

A directory password, when specified, becomes the file password. If there is no directory password, no password is required to open the file.

After the call to CreateFile, the SetFileStatus operation can be used to assign a different password to the file.

NOTE: Create is a synonym for CreateFile and may be used in its place. If, however, Create is a keyword in the language in which a program is coded, using this word will result in duplicate external references.

Procedural Interface

CreateFile (pbFileSpec, cbFileSpec, pbPassword, cbPassword, lfaFileSize):
ercType

where

pbFileSpec
cbFileSpec

describe a character string of the form {Node}[VolName] <DirName>FileName. The distinction between uppercase and lowercase in file specifications is not significant in matching file names.

pbPassword
cbPassword

describe one of the following

- a volume or directory password (if required) authorizing the creation of a file in the specified directory
- a string that is not the volume or directory password but can effectively beome the file password after a directory password is assigned
- are the value 0, meaning a volume or directory password is not required to create this file

lfaFileSize

is the file size in bytes. It must be a multiple of 512.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	5
12	reserved	2	
14	lfaFileSize	4	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	

This page intentionally left blank

CreatePartition (pbPartitionName, cbPartitionName, cParagraph, bParType, pUserNumRet): ercType

Description

CreatePartition creates a new application partition, assigns its name, and returns a user number. CreatePartition can be issued only by a process executing in an application partition.

When *bParType* is 6, CreatePartition is limited to partition-managing programs, such as the Context Manager. In this case, CreatePartition always specifies the size of the partition data structures, such as those for video and keyboard, rather than assigning default sizes.

When *bParType* is 0 or 1, CreatePartition causes the exit run file to be loaded into the application partition, replacing the application program that executed the CreatePartition operation.

Procedural Interface

CreatePartition (pbPartitionName, cbPartitionName, cParagraph, bParType, pUserNumRet): ercType

where

pbPartitionName
cbPartitionName

describe the partition name (up to 12 characters).

cParagraph

is the number of paragraphs of memory to be allocated to the application partition. If *cParagraph* is 0, then the partition is allocated all of the available memory.

If *bParType* is 6, cParagraph is the maximum number of paragraphs available to the partition and can be overridden by a smaller value.

bParType

is a byte that specifies an application's partition memory requirements including overhead for buffers, keyboard, or screen, for example.

bParType is compatible with the 0 and 0FFh values of the *fRunBatch* parameter of real mode operating systems. 0FFh indicates that a Batch Control Block of 1.3K bytes is allocated in addition to the memory for the partition itself. 0 indicates no Batch Control Block is allocated.

In addition, *bParType* has the values of 2 or 6, which are used only by partition-managing programs, such as the Context Manager. 2 and 6 indicate the creation of a partition, including virtual screen and keyboard structures with no resizing allowed upon chain; 6 allows, in addition, resizing upon chain.

pUserNumRet

is the memory address of a word where user number of the created partition is returned.

Request Block

sUserNumRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nResPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	181
12	cParagraph	2	
14	bParType	1	
15	reserved	3	
18	pbPartitionName	4	
22	cbPartitionName	2	
24	pUserNumRet	4	
28	sUserNumRet	2	2

This page intentionally left blank

CreateProcess (pProcessDescriptor): ercType

Description

The CreateProcess primitive is similar to NewProcess in that it creates a new process and schedules it for execution. The priority specified by CreateProcess may be modified by SetDeltaPriority before the process is scheduled for execution. (See SetDeltaPriority.)

CreateProcess is called by an application process to establish an application program in which multiple processes execute the same reentrant code.

CreateProcess is also called by Chain and LoadTask to create the initial process of a new program. (See "Partitions and Partition Management" in the *CTOS Operating System Concepts Manual* for more information.)

CreateProcess differs from NewProcess in that NewProcess returns additional information to the caller. (See NewProcess.)

Procedural Interface

CreateProcess (pProcessDescriptor): ercType

where

pProcessDescriptor

is the memory address of the following structure:

Offset	Field	Size (Bytes)	Description
0	pEntry	4	Memory address (CS:IP) at which to begin execution of the new process. This address is usually an operation address.
4	saData	2	Value of the segment base address to be used in accessing the new process's global data. This value is loaded into the Data Segment (DS) register when the process is scheduled for execution.*
6	saExtra	2	Value to be loaded into the Extra Segment (ES) register when the new process is scheduled for execution. This value must be a valid segment base address or 0.
8	saStack	2	Value of the segment base address of the new process's stack. This value is loaded into the Stack Segment (SS) register when the process is scheduled for execution.*

* DS and SS values must be the same in the medium model of computation, which is used by many programming languages and libraries. If the newly created process intends to share data with the calling process, the DS and SS values of the newly created and the calling processes must be the same: the current value in the calling process's DS register must be loaded into the new process's DS *and* SS registers.

Offset	Field	Size (Bytes)	Description
10	oStackInit	2	Value of the offset of the last word in a global array that is declared in the caller's data space. This value is loaded into the Stack Pointer (SP) register when the new process is scheduled for execution. The array provides memory for the newly created process to use as a stack.
12	priority	1	Priority at which the new process is to be scheduled for execution. The highest priority is 0; the lowest is 254.
13	fSys	1	Always FALSE.
14	default-Response-Exchange	2	To make operating system calls from the new proccess, allocate an exchange using the AllocExch operation and provide it here. The exchange becomes the default response exchange of the new process. The calling process must never use this exchange again to avoid possible conflict.
16	fSuspend	1	If TRUE, the process is not scheduled for execution.

* DS and SS values must be the same in the medium model of computation, which is used by many programming languages and libraries. If the newly created process intends to share data with the calling process, the DS and SS values of the newly created and the calling processes must be the same: the current value in the calling process's DS register must be loaded into the new process's DS *and* SS registers.

Request Block

CreateProcess is a Kernel primitive.

This page intentionally left blank

CreateUser (pbName, cbName, cPar, pUserNumRet, sUserStruc): ercType

Description

CreateUser creates a variable size partition. Unlike CreatePartition, CreateUser always specifies the size of the user structure (partition data structures such as those for video and keyboard) rather than assigning default sizes.

Procedural Interface

CreateUser (pbName, cbName, cPar, pUserNumRet, sUserStruc): ercType

where

pbName
cbName

describe the name (up to 12 characters) of the partition to be created.

cPar

is the maximum amount of memory (expressed in 16-byte units) available to a program running in the partition. This value may be overridden if a smaller value is specified in the header of a run file. Partitions created by CreateUser are automatically flexible in that they can change size upon chain. The partitions created by CreatePartition can be fixed or flexible in size.

pUserNumRet

is the memory address of a word where the user number of the created partition is returned.

sUserStruc

is the amount of space to be reserved for Context Manager–specific data structures. These structures include virtual video and keyboard buffers.

Request Block

sUserNumRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	296
12	cPar	2	
14	reserved	2	4
16	sUserStruc	2	
18	pbName	4	
22	cbName	2	
24	pUserNumRet	4	
28	sUserNumRet	2	2

CSubParams (iParam): WORD

Description

CSubParams returns the number of subparameters stored in the Variable Length Parameter Block for a specified parameter, that is, the number of subparameters the operator entered in a specified field of a form (such as displayed by the Executive).

Procedural Interface

CSubParams (iParam): WORD

where

iParam

is the index of the parameter.

Request Block

CSubParams is an object module procedure.

This page intentionally left blank

CurrentOsVersion (pwReleaseRet, pwRevisionRet): ercType

Description

CurrentOsVersion determines the version of an operating system by issuing a request to OsVersion. If the version of the operating system does not support the OsVersion request, CurrentOsVersion returns values of 0.

Procedural Interface

CurrentOsVersion (pwReleaseRet, pwRevisionRet): ercType

where

pwReleaseRet

is the memory address of a word to which the operating system release version is returned. For example, for operating system version 9.1, *wReleaseRet* would be 9.

pwRevisionRet

is the memory address of a word to which the operating system revision number is returned. For example, for operating system version 9.1, *wRevisionRet* would be 1.

The operating system internal version numbers and their corresponding release numbers are listed below.

CTOS and BTOS Operating Systems:

Internal Version Number	Release Numbers		
	CTOS and SRP	**XEBTOS**	**BTOS**
8.0	CTOS 8.0		BTOS 0-3.1
9.0	CTOS 9.0	XEBTOS 6.0	BTOS 4.0
9.1	CTOS 9.1		BTOS 5.0
9.5	CTOS 9.5		BTOS 6.0
9.6	CTOS 9.6		BTOS 7.0
9.7	CTOS 9.7		BTOS 8.0
9.8	CTOS 9.8		real mode BTOS II (versions less than 3.2)
9.9	CTOS 9.9		
9.10	CTOS 9.10		
9.11			real mode BTOS II 3.2
10.0	CTOS II 1.0		
10.4	CTOS/SRP 1.4	XEBTOS 7.0	
11.0	CTOS/VM 2.0		BTOS II 1.0
11.1	CTOS/VM 2.1		BTOS II 1.1
11.2	CTOS/VM 2.2		
11.3	CTOS/VM 2.3		
11.4			BTOS II 3.0
11.5			BTOS II 3.1
11.6	CTOS/VM 2.4		
11.7			protected mode BTOS II 3.2
12.0	CTOS/XE 3.0		

Merged Operating Systems:

Internal Version Number	Release Number
9.13	CTOS I 3.3
12.0	CTOS/XE 3.0
12.3	CTOS II 3.3
9.4	CTOS I 3.4
12.4	CTOS II 3.4
12.1	CTOS/EX 3.4
13.0	CTOS III 1.0

Request Block

CurrentOsVersion is an object module procedure.

This page intentionally left blank

DCXVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

DCXVersion returns a structure that identifies the version of the installed Data Communications (DCX) service.

Procedural Interface

DCXVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe a memory area to which the structure containing the version is written. The format of the structure is as follows:

Offset	Field	Size (Bytes)
0	version	2

pcbRet

describes a word into which the actual length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8227
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

DeallocAliasForServer (sgAlias, userNumOwner): ercType

Description

DeallocAliasForServer asserts that the alias selector is no longer in use and may be deallocated by the operating system.

All calls to RemakeAliasForServer must match all calls to the operation DeallocAliasForServer.

Procedural Interface

DeallocAliasForServer (sgAlias, userNumOwner): ercType

where

sgAlias

specifies the alias selector that may be deallocated.

userNumOwner

identifies the user number of the alias selector.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	337
12	reserved	2	
14	userNumOwner	2	
16	sgAlias	2	

DeallocateRods (spLow, spHigh): ercType

NOTE: This call is necessary only when a program is executing in real mode. If called by a program executing in protected mode, it does nothing and always returns status code 0 ("ercOK").

Description

DeallocateRods unwinds the stack and frees the return overlay descriptor (ROD) for evicted procedures in a program using the Virtual Code facility. A ROD identifies an overlay and the outstanding procedure in that overlay that was swapped out of memory before a return was made.

DeallocateRods must be called if a program using the Virtual Code facility deviates from the standard call/return cycle procedure, such as when local GO TO instructions are implemented using assembly language routines.

Procedural Interface

DeallocateRods (spLow, spHigh): ercType

where

spLow

is the address on the stack from which the jump begins.

spHigh

is the address on the stack at which the jump ends.

Request Block

DeallocateRods is an object module procedure.

This page intentionally left blank

DeallocateSegment (sn): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

DeallocateSegment deallocates the specified segment. The segment must have been previously allocated by the AllocateSegment operation. (See AllocateSegment.)

Procedural Interface

DeallocateSegment (sn): ercType

where

sn

is the selector (word) of the segment to be deallocated.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	435
12	sn	2	

DeallocExch (exchange): ercType

Description

DeallocExch deallocates an exchange.

Procedural Interface

DeallocExch (exchange): ercType

where

exchange

is the exchange to deallocate. This exchange must have been obtained using the AllocExch operation.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	41
12	exchange	2	

This page intentionally left blank

DeallocHugeMemory (sn): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

DeallocHugeMemory deallocates memory allocated by the AllocHugeMemory operation.

Procedural Interface

DeallocHugeMemory (sn): ercType

where

sn

is the first selector (word) that addresses the allocated memory.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	451
12	sn	2	
14	reserved	4	

DeallocMemoryLL (pSegment, cBytes): ercType

Description

DeallocMemoryLL deallocates the specified long-lived segment containing memory of the specified size. Segments must be deallocated in a sequence exactly opposite the order in which they were allocated (that is, last allocated, first deallocated).

In protected mode, this operation deallocates the segment descriptor.

Procedural Interface

DeallocMemoryLL (pSegment, cBytes): ercType

where

pSegment

is the memory address of the segment to deallocate. The offset portion must be 0. *pSegment* should be the same memory address that was returned by the corresponding AllocMemoryLL operation.

cBytes

is the size (in bytes) of the segment to deallocate. *cBytes* should be the same value that was passed to the corresponding AllocMemoryLL operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	45
12	cBytes	2	
14	pSegment	4	

DeallocMemorySL (pSegment, cBytes): ercType

Description

DeallocMemorySL deallocates the specified short-lived segment containing memory of the specified size. Segments must be deallocated in a sequence exactly opposite the order in which they were allocated (that is, the last segment allocated is the first deallocated).

In protected mode, this operation deallocates the segment descriptor.

Procedural Interface

DeallocMemorySL (pSegment, cBytes): ercType

where

pSegment

is the memory address of the segment to deallocate. The offset portion must be 0. The memory address specified by *pSegment* should be the same memory address that was returned by the corresponding AllocMemorySL operation.

cBytes

is the size (in bytes) of the segment to deallocate. The value of *cBytes* should be the same value that was passed to the corresponding AllocMemorySL operation.

In protected mode, if *cBytes* has the value 0FFFFh, the operating system determines the size of the segment to deallocate from the segment descriptor. This feature supports deallocation of static segments (such as initialization code) when segment size cannot be easily determined by the calling program.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	43
12	cBytes	2	
14	pSegment	4	

DeallocRunFile (Id): ercType

Description

DeallocRunFile deallocates code or data allocated (and loaded) using the LoadRunFile operation. DeallocRunFile also frees the selectors used to load the code or data. One or more run files can be deallocated at a time.

Procedural Interface

DeallocRunFile (Id): ercType

where

Id

is the 4 byte identification number of the run file that was returned when the the run file was loaded using the LoadRunFile operation. The run file specified by *Id* and any run files allocated and loaded later than this run file are deallocated. As an example, suppose that three run files were allocated with *Id*s in the order 1, 2, and 3, where 1 was the first allocated. If DeallocRunFile is called with *Id* 2, run file 2 and run file 3 are deallocated.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	351
12	Id	4	
16	reserved	2	

DeallocSg (sg, userNum): ercType

Caution: *This operation should be used only by programs executing in protected mode.*

Description

DeallocSg deallocates a Global Descriptor Table (GDT) selector (SG). This operation typically is used by a system service to deallocate selectors.

Procedural Interface

DeallocSg (sg, userNum): ercType

where

sg

is a GDT selector.

userNum

is the user number assigned to the GDT selector when the selector was allocated.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	345
12	sg	2	
14	userNum	2	

DeallocUserNumbers (userNumBlockId): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

Description

DeallocUserNumbers deallocates a specified block of user numbers previously allocated with the AllocUserNumbers operation. The user number block deallocated is identified by the block identification number (ID) returned by AllocUserNumbers.

Procedural Interface

DeallocUserNumbers (userNumBlockId): ercType

where

userNumBlockId

is the identification number (word) of the block of user numbers to deallocate.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	309
12	userNumBlockId	2	

DebugOp (pCmd, cbCmd, pReqBuf, cbReqBuf, pRespBuf, cbRespBuf):
ercType

Description

DebugOp performs common debugging operations. For a complete description of DebugOp, see Appendix J in the *CTOS Debugger User's Guide*.

Procedural Interface

DebugOp (pCmd, cbCmd, pReqBuf, cbReqBuf, pRespBuf, cbRespBuf):
ercType

Request Block

DebugOp is an object module procedure.

This page intentionally left blank.

DefineInterLevelStack (iLevel, pStackTop): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. It is supported only by operating systems executing in protected mode.*

Description

DefineInterLevelStack initializes the stack segment (SS) and the stack pointer (SP) fields of the caller's Task State Segment for the protection level specified by *iLevel*. *iLevel* must be 0, 1 or 2. The selector portion of *pStackTop* is assigned to the SS field and the offset portion is assigned to the SP field.

DefineInterLevelStack supports system software, such as the PC Emulator, which operates at more than one protection level.

For details on protected mode, see the following Intel manuals:

- *iAPX Programmer's Reference Manual*
- *80386 Programmer's Reference Manual*

Procedural Interface

DefineInterLevelStack (iLevel, pStackTop): ercType

where

iLevel

is the protection level. *iLevel* must be 0, 1, or 2.

pStackTop

is the memory address of the top of the stack for the specified protection level.

Request Block

DefineInterLevelStack is a system-common procedure.

DefineLocalPageMap (pFrames, saLocal, cFrames): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. It is supported only by operating systems executing in protected mode.*

Description

DefineLocalPageMap defines an address mapping between the linear address specified by *saLocal* and the physical address referenced by *pFrames*. Both *saLocal* and *pFrames* must be aligned on a 4K-byte boundary. The mapping extends for the number of frames specified by *cFrames*.

The AllocMemoryFramesSL operation allocates memory with the required alignment.

DefineLocalPageMap supports system software, such as the PC Emulator, which creates and maintains page maps on the 80386 microprocessor.

For details on page maps, see the (Intel) *80386 Programmer's Reference Manual.*

Procedural Interface

DefineLocalPageMap (pFrames, saLocal, cFrames): ercType

where

pFrames

is the memory address of the page frames.

saLocal

is the linear address.

cFrames

is the number of specified 4K-byte frames.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	8
2	nReqPbCb	1	0
3	nRespPbCp	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	320
12	pFrames	4	
16	saLocal	2	
18	cFrames	2	

DeinstallQueueManager (pUserNumRet): ercType

Description

DeinstallQueueManager provides a method of terminating operation of the Queue Manager and freeing its memory partition. It shuts down all operations and unlocks its memory partition. The caller can then vacate and remove the partition.

Procedural Interface

DeinstallQueueManager (pUserNumRet): ercType

where

pUserNumRet

is a memory address of a word to which the Queue Manager user number is returned.

Request Block

sUserNumRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32867
12	pUserNumRet	4	
16	sUserNumRet	2	2

DeinstPSServer (pPSUserNumRet): ercType

Description

DeinstPSServer is a synonym for PSDeinstServer. See the description of PSDeinstServer for details.

This page intentionally left blank

Delay (n): ercType

Description

Delay suspends the execution of the client for the specified interval.

Procedural Interface

Delay (n): ercType

where

n

is the interval to delay (in units of 100 milliseconds).

Request Block

Delay is a system-common procedure.

This page intentionally left blank

DeleteByteStream (pBswa): ercType

Description

DeleteByteStream deletes the file associatedd with the open byte stream identified by the memory address of the Byte Stream Work Area.

Procedural Interface

DeleteByteStream (pBswa): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

Request Block

DeleteByteStream is an object module procedure.

This page intentionally left blank

DeleteDaRecord (pDawa, qiRecord): ercType

Description

DeleteDaRecord deletes a record from the open DAM file identified by the memory address of the Direct Access Work Area (DAWA). The deleted record is specified by the record number. Once a record is deleted, it can no longer be read.

Procedural Interface

DeleteDaRecord (pDawa, qiRecord): ercType

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord

is a 32-bit unsigned integer specifying the number of the record to be deleted.

Request Block

DeleteDaRecord is an object module procedure.

This page intentionally left blank

DeleteDir (pbDirSpec, cbDirSpec, pbPassword, cbPassword): ercType

Description

DeleteDir deletes an empty directory. A program cannot use this operation until it has deleted all files in the directory by using the Executive **Delete** command or by using the DeleteFile operation.

Status code 241 ("Directory not empty") is returned if the directory is not empty.

Procedural Interface

DeleteDir (pbDirSpec, cbDirSpec, pbPassword, cbPassword): ercType

where

pbDirSpec
cbDirSpec

describe a character string of the form {Node}[VolName]DirName.

pbPassword
cbPassword

describe the volume or directory password. *Either* password authorizes the deletion of the directory.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	18
12	reserved	6	
18	pbDirSpec	4	
22	cbDirSpec	2	
24	pbPassword	4	
28	cbPassword	2	

DeleteFile (fh): ercType

Description

DeleteFile deletes an open file.

NOTE: Delete is a synonym for DeleteFile and may be used in its place. If, however, Delete is a keyword in the language in which a program is coded, using this word will result in duplicate external references.

Procedural Interface

DeleteFile (fh): ercType

where

fh

is the file handle returned from an OpenFile operation. The file must be open in modify mode.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	6
12	fh	2	

DeviceInService (pTyDevRet): ercType

Caution: This operation is for use only by programs executing in protected mode.

Description

The DeviceInService Kernel primitive returns the type of device being interrupted in a device-independent manner. The advantage of using this operation is that the calling program does not need to provide hardware details of a particular programmable interrupt controller to determine which device is being interrupted. For example, the caller does not need to know the scheme for nesting interrupts generated by devices of different priorities. A program using this operation can be ported to other systems with different hardware requirements.

Procedural Interface

DeviceInService (pTyDevRet): ercType

where

pTyDevRet

is the memory address at which the type of the device (word) is returned. For a list of the device types and their values, see the description of the ControlInterrupt operation. If the value 0 is returned, the caller is not in an interrupt handler.

Request Block

DeviceInService is a Kernel primitive.

This page intentionally left blank

DisableActionFinish (fDisable): ercType

Description

DisableActionFinish permits an application program in an application partition to disable operating system interpretation of the ACTION-FINISH key combination.

Normally, the operator can terminate the current application program by simultaneously pressing the ACTION and the FINISH keys. However, it is highly undesirable to terminate an application that is executing a critical section of code, such as when a system service is installing but has not yet called ConvertToSys.

Programs should use ReadActionKbd in conjunction with DisableActionFinish to determine whether or not the user pressed ACTION-FINISH during execution of a critical section. If the user pressed ACTION-FINISH, the program should inform the user that he cannot terminate until execution of the critical section is complete.

Procedural Interface

DisableActionFinish (fDisable): ercType

where

fDisable

disables ACTION-FINISH if TRUE or enables ACTION-FINISH if FALSE.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	67
12	fDisable	2	

DisableCluster (fDisablePoll, timeInterval): ercType

Description

DisableCluster allows an application program on the server to disable polling of the cluster workstations after a specified time period. DisableCluster is also used to resume polling of the cluster workstations.

During the specified time period, the GetDateTime operation returns status code 46 ("Server going down") and the time left (in seconds) before polling stops. After the specified time period, all operations return status code 6 ("Server not running").

Typically, the application program (such as the Executive) on the cluster workstation that performs the GetDateTime operation would notify the cluster workstation user when it received the "Server going down" status code.

DisableCluster is useful for stopping all cluster workstation activity, for example, to perform software maintenance on the server.

Procedural Interface

DisableCluster (fDisablePoll, timeInterval): ercType

where

fDisablePoll

disables polling if TRUE or resumes it if FALSE.

timeInterval

is the time period (in tenths of a second). This is not meaningful if *fDisablePoll* is FALSE.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	122
12	fDisablePoll	2	
14	timeInterval	2	

DiscardInputBsC (pBs, pCbRet): ercType

Description

DiscardInputBsC discards any characters in the receive queue.

(See "Communications Programming" in the *CTOS Operating System Concepts Manual.)*

Procedural Interface

DiscardInputBsC (pBs, pCbRet): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

pCbRet

is the memory address to which the count of the number of characters discarded is returned (word).

Request Block

DiscardInputBsC is an object module procedure.

This page intentionally left blank

DiscardOutputBsC (pBs): ercType

Description

DiscardOutputBsC discards any characters in the transmit queue. This procedure is useful with printers, to abort printing.

(See "Communications Programming" in the *CTOS Operating System Concepts Manual.)*

Procedural Interface

DiscardOutputBsC (pBs): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

Request Block

DiscardOutputBsC is an object module procedure.

This page intentionally left blank

DismountVolume (pbVolName, cbVolName, pbPassword, cbPassword):
ercType

Description

DismountVolume dismounts the specified volume.

Dismounting (and mounting) of volumes is normally controlled by the Automatic Volume Recognition (AVR) capability of the file management system. The Dismount (and Mount) operations are provided for the use of commands, such as **Format Disk**, that must override AVR. (See the *CTOS Executive Reference Manual* for a description of the **Format Disk** command.)

Procedural Interface

DismountVolume (pbVolName, cbVolName, pbPassword, cbPassword):
ercType

where

pbVolName
cbVolName

describe a character string of the form {Node}[VolName]. Square brackets are optional for the device name. The distinction between uppercase and lowercase is not significant in matching device names.

pbPassword
cbPassword

describe the volume password that authorizes access to the specified volume.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	12
12	reserved	6	
18	pbVolName	4	
22	cbVolName	2	
24	pbPassword	4	
28	cbPassword	2	

DmaMapBuffer (sn, ra32, cb32, wDmaFence, pPaRet, pCb32Ret, fUninitialized, bus): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

DmaMapBuffer prepares an area of linear address space for DMA use by mapping the linear address space to a contiguous, locked-in area in physical memory. If the linear buffer lacks a contiguous, adequately sized physical address space, DmaMapBuffer searches for such an area in unused physical memory. It then maps the address of the area as the physical address of the buffer. If no adequate free memory space can be found, DmaMapBuffer constructs one by swapping out pages.

To allow DMA hardware to read the buffer, DmaMapBuffer optionally preserves the contents of the buffer during construction or relocation. Specifically, it copies data from non-contiguous to contiguous frames and faults in not-present pages of the DMA buffer.

The caller must provide DmaMapBuffer with the logical address of the DMA buffer. DmaMapBuffer then returns the designated physical address of the buffer, as well as the amount of data that could be successfully mapped.

By mapping data to a contiguous, locked-in memory area, DmaMapBuffer provides a buffer that can be accessed by DMA hardware in a single read/write operation.

On some types of hardware (for example, SuperGen Series 5000 workstations with X-Bus adapters), the value returned by DmaMapBuffer may be a bus address rather than a true physical address.

Procedural Interface

DmaMapBuffer (sn, ra32, cb32, wDmaFence, pPaRet, pCb32Ret, fUninitialized, bus): ercType

where

sn

is the segment name (selector) of the DMA buffer. The selector can be a GDT selector or an LDT selector.

ra32

is the 32-bit segment offset of the DMA buffer.

cb32

is a double word that specifies the size of the DMA buffer.

wDmaFence

is a word indicating the existence of fences in physical memory that cannot be crossed by the DMA operation. Some DMA hardware will not increment past certain boundaries. A value of 0 indicates no boundary restrictions. A value of 1 indicates that a 64K boundary should not be crossed; a value of 2 indicates that a 128K boundary should not be crossed, and so forth.

pPaRet

is the memory address of a double word to which the physical address of the DMA buffer is returned. If mapping hardware is present, this value may be a bus address rather than a true physical address.

pCb32Ret

is the memory address of a double word to which the number of bytes successfully mapped is returned. Due to limits on the amount of data transferrable in a single DMA operation, or due to a lack of available contiguous page frames, this value may be less than the size of the DMA buffer.

fUninitialized

indicates, when TRUE, that the DMA buffer contents need not be preserved during construction of a contiguous buffer. Note that the contents can be discarded if the DMA hardware will write data to the buffer.

bus

identifies the type of module that will access the DMA buffer. This parameter allows the operating system to set up DMA mapping tables, if necessary. DmaMapBuffer must be called each time a different type of module accesses the DMA buffer. Each value identifies the following module type:

Value	Module Type
0	Main processor
1	Reserved
2	X-Bus module
3	X-Bus+ cartridge

Request Block

sPaRet and *sCb32Ret* are always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	16
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	436
12	reserved	2	
14	ra32	4	
18	cb32	4	
22	wDmaFence	2	
24	fUninitialized	2	
26	bus	2	
28	reserved	2	
30	sn	2	
32	reserved	2	1
34	pPaRet	4	
38	sPaRet	2	4
40	pCb32Ret	4	
44	sCb32Ret	2	4

NOTE: This operation has been deleted in Update Notice 1, August, 1992.

Deleted with Update Notice 1, August, 1992.

Deleted with Update Notice 1, August, 1992.

This page intentionally left blank.

DmaUnmapBuffer (sn, ra32, cb32, pa, bus): ercType

NOTE: This operation is for use only by programs executing in protected mode.

Description

DmaUnmapBuffer informs the operating system that the specified linear addresses are no longer in use as a DMA buffer. Once notified, the operating system is free to use the underlying physical memory area of the buffer for other purposes.

An application calls DmaUnmapBuffer to unmap addresses that were previously mapped by the DmaMapBuffer operations. (See DmaMapBuffer.)

Procedural Interface

DmaUnmapBuffer (sn, ra32, cb32, pa, bus): ercType

where

sn

is the segment name (selector) of the DMA buffer.

ra32

is the 32-bit segment offset of the DMA buffer.

cb32

is the 32-bit value specifying the size of the DMA buffer.

pa

is the 32-bit physical address that was returned by a previous call to DmaMapBuffer.

bus

identifies the module where the DMA buffer is located. This parameter allows the operating system to release DMA mapping tables, if necessary. Each value indicates the following:

Value	Description
0	Main processor
1	Reserved
2	X-Bus module
3	X-Bus+ cartridge

Request Block

DmaUnmapBuffer is a system-common procedure.

DoDirectRead (fh, pbCmd, cbCmd, pBuffer, sBuffer, pbResult, cbResult, wOptions, sDmaXfer): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. It is supported only to provide compatibility with older operating system versions that support SCSI devices but not the SCSI Manager. New applications are encouraged to use the SCSI Manager operations.*

Description

DoDirectRead issues a command to a SCSI device for which a Device Control Block has been allocated (generally, direct-access devices only) and provides an optional DATA IN phase for the transfer of data from the device to memory.

Procedural Interface

DoDirectRead (fh, pbCmd, cbCmd, pBuffer, sBuffer, pbResult, cbResult, wOptions, sDmaXfer): ercType

where

fh

identifies the peripheral device.

pbCmd
cbCmd

describe the string of command bytes to write to the peripheral device.

pBuffer
sBuffer

describe the buffer into which data is read by DMA.

pbResult
cbResult

describe the buffer into which command result bytes are written.

wOptions

specifies periperal specific options.

sDmaXfer

is the DMA transfer byte count used to program the DMA channel.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	318
12	fh	2	
14	wOptions	2	
16	sDmaXfer	2	
18	pbCmd	4	
22	cbCmd	2	
24	pBuffer	4	
28	sBuffer	2	
30	pbResult	4	
34	cbResult	2	

This page intentionally left blank

DoDirectWrite (fh, pbCmd, cbCmd, pBuffer, sBuffer, pbResult, cbResult, wOptions, sDmaXfer): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. It is supported only to provide compatibility with older operating system versions that support SCSI devices but not the SCSI Manager. New applications are encouraged to use the SCSI Manager operations.*

Description

DoDirectWrite issues a command to a SCSI device for which a Device Control Block has been allocated (generally, direct-access devices only) and provides an optional DATA OUT phase for the transfer of data from memory to the device.

Procedural Interface

DoDirectWrite (fh, pbCmd, cbCmd, pBuffer, sBuffer, pbResult, cbResult, wOptions, sDmaXfer): ercType

where

fh

identifies the peripheral device.

pbCmd
cbCmd

describe the string of command bytes to write to the peripheral device.

pBuffer
sBuffer

describe the buffer from which data is written by DMA.

pbResult
cbResult

describe the buffer into which command result bytes are written.

wOptions

specifies peripcral specific options.

sDmaXfer

is the DMA transfer byte count used to program the DMA channel.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	319
12	fh	2	
14	wOptions	2	
16	sDmaXfer	2	
18	pbCmd	4	
22	cbCmd	2	
24	pBuffer	4	
28	sBuffer	2	
30	pbResult	4	
34	cbResult	2	

This page intentionally left blank

Doze (wTime): ercType

Description

Doze, like the Delay operation, suspends the execution of the client for the specified interval. Unlike Delay, Doze allows timing to continue while the caller is swapped out of memory. If the time interval elapses while the caller is swapped out, the caller is automatically swapped back into memory and resumes execution. This provides an accurate time delay without disabling swapping.

Doze is used by the **Submit** command to start executing a submit file at a user-specified time. (For details on **Submit**, see the *CTOS Executive Reference Manual*.)

Procedural Interface

Doze (wTime): ercType

where

wTime

is the time interval (word) to delay (in units of 100 milliseconds).

Request Block

Doze is a system-common procedure.

This page intentionally left blank

DrainTerminalOutput has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *This operation works on shared resource processor hardware only. Do not use it on workstation hardware.*

Description

DrainTerminalOutput is issued by the client to ensure an empty output buffer. DrainTerminalOutput returns only after all output has been flushed. This operation is commonly used prior to a call to SetTerminal.

Procedural Interface

DrainTerminalOutput has no procedural interface. You must make a request block and issue the request using one of the request operations listed above

where

bDestCPU

is the slot number of the Cluster or Terminal Processor that is to handle the request.

bSourceCPU

is the client's slot number.

bPort

refers to the physical port number of a terminal attached to an RS-232-C port.

bWindow

must be 0.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	2	4
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	ExchResp	2	
8	ercRet	2	
10	rqCode	2	12313
12	bDestCPU	1	
13	bSourceCPU	1	
14	bPort	1	
15	bWindow	1	

EnableSwapperOptions (pProc, fFixupStubs): ercType

Caution: *In protected mode, this operation returns status code 7 ("Not implemented"). It is recommended, therefore, that you run your program in real mode while using this operation to optimize overlays. Then, delete this operation and relink to run your program in protected mode.*

Description

EnableSwapperOptions allows an arbitrary procedure to be called each time the OverlayFault operation is called in a program using the Virtual Code facility.

If *fFixupStubs* is FALSE, every call to any overlay procedure passes through OverlayFault. Thus, every call can be logged by a user-written operation.

You can use this operation to optimize the contents of your overlays by gathering statistics on how often various overlay routines are called.

Procedural Interface

EnableSwapperOptions (pProc, fFixupStubs): ercType

where

pProc

is the memory address of the procedure to be called at each invocation of OverlayFault. This user-written procedure is called with two word parameters, *iOv* and *iStub*, which are respectively the overlay number and the stub of the operation being called.

fFixupStubs

is a flag indicating whether or not the memory addresses of the stubs should be changed to reflect the actual memory addresses of operations in swapped-in overlays. If this flag is TRUE, the Virtual Code facility operates normally, changing the stubs. If it is FALSE, all calls continue to go to the OverlayFault operation.

Request Block

EnableSwapperOptions is an object module procedure.

EnlsAppendChar (pbString, cbString, sBuffer, pChar, pcbStringRet): ercType

NOTE: An application should use EnlsAppendChar and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsAppendChar appends a single-byte or multibyte character to a string of characters. Upon returning, EnlsAppendChar provides the length of the resulting string.

EnlsAppendChar is often used to modify text strings that appear in a video display. Initially, the application calls QueryFrameChar to obtain a copy of the string. After it has used EnlsAppendChar, the application then calls PutFrameChar to display the modified string. (See QueryFrameChar and PutFrameChar.)

Status code 15202 ("Buffer full") is returned if the buffer is not large enough to hold the resulting string.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsAppendChar (pbString, cbString, sBuffer, pChar, pcbStringRet): ercType

where

pbString
cbString

describe the character string to which the character is to be appended.

sBuffer

is the size (in bytes) of the buffer in which the character string is located.

pChar

is the memory address of the character to be appended. The character may be one to four bytes in size.

pcbStringRet

is the memory address of a word where the length of the resulting character string is returned.

Request Block

EnlsAppendChar is an object module procedure in Enls.lib.

EnlsCase (pNlsTableArea, pCharIn, pCharOutRet, fUpper): ercType

NOTE: An application should use EnlsCase and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsCase converts a given character from lowercase to uppercase, or from uppercase to lowercase. If it receives a single-byte character, EnlsCase in turn calls NlsCase. If, however, EnlsCase receives a multibyte character, it uses its own code to convert the character.

Unlike the NlsCase operation, EnlsCase converts both single-byte and multibyte characters. For this reason, EnlsCase should be used in place of NlsCase.

Status code 15203 ("Bad character") is returned if the specified character is a null value.

For details on Extended Native Language Support, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsCase (pNlsTableArea, pCharIn, pCharOutRet, fUpper): ercType

where

pNlsTableArea

is the memory address of the NLS table to be used. EnlsCase returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the currently loaded character set.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Lowercase-to-Uppercase or Uppercase-to-Lowercase table.
- If *pNlsTableArea* is the address of an alternate set of NLS tables, an answer is returned based on the NLS Lowercase-to-Uppercase or Uppercase-to-Lowercase table in the alternate set of NLS tables.

pCharIn

is the memory address of a double word which contains the character to be converted.

pCharOutRet

is the memory address of a double word to which the converted character is returned.

fUpper

is a flag which, when TRUE, causes EnlsCase to perform a lowercase to uppercase conversion. When FALSE, it causes EnlsCase to perform an uppercase to lowercase conversion.

Request Block

EnlsCase is an object module procedure in Enls.lib.

EnlsCbToCCols (pbString, cbString, pcColsRet): ercType

NOTE: An application should use EnlsCbToCCols and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsCbToCCols ('Count of Bytes to Count of Columns') returns the number of screen or printer columns needed to display a string of single-byte or multibyte characters.

Typically, EnlsCbToCCols is used with the PutFrameChars and PosFrameCursor operations. After the application uses PutFrameChars to write text to the screen, it often places the cursor at the end of the text string. Before it can position the cursor, however, the application must first use EnlsCbToCCols to determine the number of columns that the text string occupies. To move the cursor, the application can then call PosFrameCursor, specifying the number of columns returned by EnlsCbToCCols. (See PosFrameCursor.)

Status code 15208 ("Bad string size") is returned if the specified string size is a negative value.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsCbToCCols (pbString, cbString, pcColsRet): ercType

where

pbString
cbString

describe the character string for which the number of columns will be returned.

pcColsRet

is the memory address of a word where the number of columns required to display or print the character string is returned.

Request Block

EnlsCbToCCols is an object module procedure in Enls.lib.

EnlsClass (pNlsTableArea, pCharIn, pbClassReturn): ercType

NOTE: An application should use EnlsClass and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsClass returns the class of a single-byte or multibyte character. A class is defined as one of six categories: numeric, alpha, special, graphic, blind, or other text.

If it receives a single-byte character, EnlsClass in turn calls NlsClass. If EnlsClass receives a multibyte character, it uses its own code to classify the character.

EnlsClass, unlike NlsClass, can classify either single-byte or multibyte characters. For this reason, EnlsClass should be used in place of NlsClass.

An application can use EnlsClass to prevent inappropriate characters from being processed. For example, if a command field is to be filled with only a file name, the caller can use EnlsClass to identify graphic or other inappropriate typed-in characters.

Note that EnlsClass uses the NLS table to return the classes of single-byte characters. Alternately, EnlsClass uses the currently loaded character set to return the classes of multibyte characters.

Status code 15203 ("Bad character") is returned if the specified character code cannot be classified.

For details on Extended Native Language Support, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsClass (pNlsTableArea, pCharIn, pbClassReturn): ercType

where

pNlsTableArea

is the memory address of the NLS Character Class table. EnlsClass returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the currently loaded character set.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Character Class table.
- If *pNlsTableArea* is the address of an alternate set of NLS tables, an answer is returned based on the NLS Character Class table in the alternate set of NLS tables.

pCharIn

is the memory address of a double word that contains the character to be classified.

pbClassReturn

is the memory address of a byte where the class is returned. Each value represents the following class:

Value	Class	Examples
0	Numeric	0, 1, or 2.
1	Alpha	A...Z, a...z

Value	Class	Examples
2	Special	¢, ½, ¼, !, ", #, $...
3	Graph	Line drawing or other graphical characters
4	Blind	Characters generally not intended for display, such as **Copy** or **Return**
5	Other Text	Includes multibyte characters
6–255	reserved	Reserved for other characters

Request Block

EnlsClass is an object module procedure in Enls.lib.

This page intentionally left blank

EnlsDeleteChar (pbString, cbString, oChar, pcbStringRet): ercType

NOTE: An application should use EnlsDeleteChar and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsDeleteChar deletes a single-byte or multibyte character from the specified character string. The application provides the offset of the first byte of the character to be deleted, and EnlsDeleteChar returns the length of the resulting string.

EnlsDeleteChar is often used to modify text strings that appear in a video display. Initially, the application calls QueryFrameChar to obtain a copy of the string. After it has used EnlsDeleteChar, the application then calls PutFrameChars to display the modified string. (See QueryFrameChar and PutFrameChars.)

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsDeleteChar (pbString, cbString, oChar, pcbStringRet): ercType

where

pbString
cbString

describe the character string from which a character is to be deleted.

oChar

is the offset, from the beginning of the character string, of the character to be deleted.

pcbStringRet

is the memory address of a word where the length of the resulting character string is returned.

Request Block

EnlsDeleteChar is an object module procedure in Enls.lib.

EnlsDrawBox (iFrame, iULCol, IULLine, iLRCol, iLRLine, bLineType, bAttr): ercType

NOTE: An application should use EnlsDrawBox and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsDrawBox uses single-byte or multibyte characters to draw a box to a video display. The application specifies the box's upper-left and lower-right corners, as well as the video frame where it will be displayed. In addition, the application specifies the attributes of the lines that will form the box.

To draw a box, an application should use EnlsDrawBox in place of the PutFrameChars and FillFrame operations. Unlike the latter operations, EnlsDrawBox does not require application-supplied strings of single-byte or multibyte characters.

Status code 15201 ("Invalid line type") is returned if the value specified in the *bLineType* parameter is invalid.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsDrawBox (iFrame, iULCol, IULLine, iLRCol, iLRLine, bLineType, bAttr): ercType

where

iFrame

is the video frame in which to draw the box.

iULCol

is the column where the upper-left corner of the box will be located.

iULLine

is the line (row) where the upper-left corner of the box will be located.

iLRCol

is the column where the lower-right corner of the box will be located.

iLRLine

is the line (row) where the lower-right corner of the box will be located.

bLineType

is the type of line used to draw the box. Each value has the following meaning:

Value	Description
0	Single, thin line
1	Single, thick line
2	Double, thin line

bAttr

is the attribute to be applied to the line. This parameter is equivalent to the *bAttr* parameter used by the PutFrameAttrs operation. Each bit position, when set, indicates the following:

Bit Position	Attribute
0	Half-bright
1	Underline
2	Reverse video
3	Blinking
4	Bold
5	Struck-through

Request Block

EnlsDrawBox is an object module procedure in Enls.lib.

This page intentionally left blank

EnlsDrawFormChars (iFrame, iCol, iLine, pbFormChars, cbFormChars, pbAttrs, cbAttrs): ercType

NOTE: An application should use EnlsDrawFormChars and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsDrawFormChars displays a string of single-byte or multibyte form characters at a specified location on the video display. The application provides a string of standard form character codes. To display the string, EnlsDrawFormChars then uses the corresponding codes from the currently loaded character set.

For example, suppose a Japanese character set is currently installed. The application would specify a code from the standard character set (for example, CEh), and EnlsDrawFormChars would use the matching code from the Japanese character set (in this case, A0h CEh).

To draw a string of form characters, an application should use EnlsDrawFormChars instead of PutFrameCharsAndAttrs. Unlike PutFrameCharsAndAttrs, EnlsDrawFormChars does not require the application to specify the correct character codes for different character sets.

Status code 15215 ("Bad form character size") is returned if *cbFormChars* is a negative value.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsDrawFormChars (iFrame, iCol, iLine, pbFormChars, cbFormChars, pbAttrs, cbAttrs): ercType

where

iFrame

is the video frame in which to write the line-drawing characters.

iCol

is the column in which to write the line-drawing characters.

iLine

is the line (row) in which to write the line-drawing characters.

pbFormChars
cbFormChars

describe the string of line-drawing characters to be written to the video display.

pbAttrs
cbAttrs

describe a string of attribute bytes to be written to the screen along with the line-drawing characters.

Request Block

EnlsDrawFormChars is an object module procedure in Enls.lib.

EnlsDrawLine (iFrame, iCol, iLine, cCols, bLineType, bAttr): ercType

NOTE: An application should use EnlsDrawLine and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsDrawLine uses single-byte or multibyte characters to draw a line on the video display. The application specifies the frame, column, and line (row) in which the line is to begin. In addition, the application specifies the type and attribute of the line.

To draw a line, an application should use EnlsDrawLine instead of the PutFrameChars and FillFrame operations. Unlike the latter operations, EnlsDrawLine does not require an application-supplied string of single-byte or multibyte line-drawing characters.

Status code 15201 ("Invalid line type") is returned if the value specified in the *bLineType* parameter is invalid.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsDrawLine (iFrame, iCol, iLine, cCols, bLineType, bAttr): ercType

where

iFrame

is the video frame in which to draw the line.

iCol

is the starting column of the line to be drawn.

iLine

is the starting row of the line to be drawn.

cCols

is the number of columns in which to draw the line.

bLineType

is the type of line to be drawn. Each value indicates the following:

Value	Description
0	Draw a single thin line.
1	Draw a single thick line.
2	Draw a double thin line.

bAttr

is the attribute to be applied to the line. This parameter is equivalent to the *bAttr* parameter used by the PutFrameAttrs operation. Each bit position, when set, indicates the following line attribute:

Bit Position	Attribute
0	Half-bright
1	Underline
2	Reverse video
3	Blinking
4	Bold
5	Struck-through

Request Block

EnlsDrawLine is an object module procedure in Enls.lib.

This page intentionally left blank

EnlsFieldEdit (iFrame, iCol, iLine, pEditState): ercType

NOTE: An application should use EnlsFieldEdit and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code. For details on Extended Native Language Support, see "Native Language Support" in the CTOS Operating System Concepts Manual.

Description

EnlsFieldEdit edits a string of characters on the video display. Input to the string is derived from characters typed at the keyboard. The characters typed can be keys defining edit commands, keys that terminate the editing session, or character data to be incorporated into the string.

Editing is guided by an edit structure containing information such as the location and size of the string to be edited, whether or not to insert characters, and the cursor position.

EnlsFieldEdit should be used instead of the FieldEdit operation. Unlike the latter, EnlsFieldEdit can edit strings that contain either single-byte or multibyte characters.

When an exit key is pressed, EnlsFieldEdit terminates and returns status code 15204 ("Exit character typed"). The following are accepted exit keys for standard keyboards: **Return**, **Next**, **Cancel**, **Help**, **Up arrow**, **Down arrow**, **Finish**, and **Go**.

The following keys define edit commands:

Key	Command
Backspace	Move cursor one space backwards
Left arrow	Backspace without deleting the character
Shift-Left arrow	Move cursor left fast
Code-Left arrow	Move cursor to beginning of string
Right arrow	Move cursor right
Shift-Right arrow	Move cursor right fast
Code-Right arrow	Move cursor to end of string
Delete	Delete character at current cursor position
Code-Delete	Delete string at current cursor position
Overtype (toggled)	Overwrite text with retrieved characters at current cursor position
Overtype (untoggled)	Insert but do not overwrite text with retrieved characters at current cursor position

Procedural Interface

EnlsFieldEdit (iFrame, iCol, iLine, pEditState): ercType

where

iFrame

is the video frame of the string to be edited.

iCol

is the starting screen column of the string to be edited.

iLine

is the starting screen line (row) of the string to be edited.

pEditState

is the memory address of the Edit State structure. In this structure, the caller places information about the string. EnlsFieldEdit, in turn, returns the final state of the string and its field in the same structure.

The *chExit*, *fAutoExited*, and *fModified* fields are set exclusively by EnlsFieldEdit. The caller sets *oCursor* with the inital cursor position, while EnlsFieldEdit sets *oCursor* with the final cursor position after editing is completed. The *fInsertLiteral* field is also set by both the caller and EnlsFieldEdit. All other fields are set only by the caller. The format of the Edit State structure is shown below.

Offset	Field	Size (Bytes)	Description
0	pbString	4	The pointer to the string of characters to be edited.
4	cbString	2	The size (in bytes) of the character string to be edited.
6	cbStringMax	2	The size (in bytes) of the memory area where the string resides. It is also the maximum number of character bytes that can be placed in the buffer in which the string resides. *cbStringMax* must be equal to or greater than *cbFieldMax*.

Offset	Field	Size (Bytes)	Description
8	cbFieldMax	2	The maximum number of bytes that can fit into the display field which is being edited.
10	oCursor	2	The offset of the cursor within the string.
12	oDisplay	2	The string offset of the first character to be displayed in the display field.
14	cbExit	2	The count of bytes of the exit character string.
16	pbExit	4	The memory address of the string of allowable exit characters.
20	chExit	4	A double word which contains the character that caused the exit.
24	bFieldType	1	The type of display field to be edited. Each value has the following meaning:

Value	Description
0	A standard field.
1	A password field (echo '#' instead of characters.)
2	A file specification field (echo '#' after a caret.)

Offset	Field	Size (Bytes)	Description
25	fAutoExit	1	A flag that, when TRUE, causes EnlsFieldEdit to return control to the caller when the cursor reaches the end of the display field.
26	fInsertLiteral	1	If the caller sets this flag to TRUE, the next keystroke is treated as a literal keystroke. If the returned value of *fInsertLiteral* is TRUE, a literal keystroke was entered into the string.
27	fAutoExited	1	A flag which, when TRUE, indicates that the cursor reached the end of the field, causing EnlsFieldEdit to return control to the caller.
28	fModified	1	A flag which, when TRUE, indicates that the user modified the display field.
29	empty	1	Reserved.

Offset	Field	Size (Bytes)	Description
30	bDisplayType	1	The edit mode. Each value indicates the following:

Value	Description
0	Edit the string and display the result.
1	Display the initial string, but do not update it on the video display or edit it in memory.
2	Edit the string but do not display initial string or editing changes.

Request Block

EnlsFieldEdit is an object module procedure in Enls.lib.

EnlsFieldEditByChar (iFrame, iCol, iLine, pChar, pEditState): ercType

NOTE: An application should use EnlsFieldEditByChar and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code. For details on Extended Native Language Support, see the CTOS Operating System Concepts Manual.

Description

EnlsFieldEditByChar edits a string of characters on the video display. Input to the string is derived from characters provided by the caller, one character at a time. The supplied character can either be an edit command or a printable character. If the character is printable, EnlsFieldEditByChar incorporates it into the string.

The editing performed by this operation is guided by an edit structure in the same format as the one used by EnlsFieldEdit. (See the description of EnlsFieldEdit for more information.)

EnlsFieldEditByChar differs from EnlsFieldEdit in that it receives a character from a caller-supplied memory address rather than the keyboard. In addition, unlike the latter operation, EnlsFieldEditByChar always returns after it processes a single character.

The following keys define edit commands:

Key	Command
Backspace	Move cursor one space backwards
Left arrow	Backspace without deleting the character
Shift–Left arrow	Move cursor left fast

Key	Command
Code–Left arrow	Move cursor to beginning of string
Right arrow	Move cursor right
Shift–Right arrow	Move cursor right fast
Code–Right arrow	Move cursor to end of string
Delete	Delete character at current cursor position
Code–Delete	Delete string at current cursor position
Overtype (toggled)	Overwrite text with returned characters at current cursor position
Overtype (untoggled)	Insert but do not overwrite text with returned characters at current cursor position

Procedural Interface

EnlsFieldEditByChar (iFrame, iCol, iLine, pChar, pEditState): ercType

where

iFrame

is the video frame which contains the string to be edited.

iCol

is the starting screen column of the string to be edited.

iLine

is the starting screen line (row) of the string to be edited.

pChar

is the memory address of the character code. This character code may be one to four bytes in size.

pEditState

is the memory address of the Edit State structure. In this structure, the calling program places information about the string that is located on the video display. EnlsFieldEditByChar, in turn, returns the final state of the string and the edit field in this structure. (For a description of the Edit State structure, see the *pEditState* parameter of EnlsFieldEdit.)

Request Block

EnlsFieldEditByChar is an object module procedure in Enls.lib.

This page intentionally left blank

EnlsFindC (pbString, pChar, cbString, pwOffsetRet): ercType

NOTE: An application should use EnlsFindC and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications tc process single-byte and multibyte characters without changes to the program code.

Description

EnlsFindC searches a string for a specified single or multibyte character and returns the string offset at which the character is found.

EnlsFindC can be used with the EnlsDeleteChar operation. After the application has found the location of a specific character with EnlsFindC, it can call EnlsDeleteChar to delete the character. (See EnlsDcleteChar.)

To search strings for characters, an application should use EnlsFindC instead of high-level language procedures (for example, the FindB procedure in PL/M).

EnlsFindC differs from EnlsFindRC in that the latter operation searches in reverse order for a character. (See EnlsFindRC.)

The value 65535 is returned at *pwOffsetRet* if the specified character is not found. Status code 15208 ("Bad string size") is returned if the specified string size is a negative value.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsFindC (pbString, pChar, cbString, pwOffsetRet): ercType

where

pbString

is the memory address where the character string is located.

pChar

is the memory address of the character whose offset is to be returned. This character may be one to four bytes in size.

cbString

is the size of the character string at *pbString*.

pwOffsetRet

is the memory address of a word where the character's offset is returned.

Request Block

EnlsFindC is an object module procedure in Enls.lib.

EnlsFindRC (pbString, pChar, cbString, pwOffsetRet): ercType

NOTE: An application should use EnlsFindRC and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code. For details on Extended Native Language Support, see the CTOS Operating System Concepts Manual.

Description

EnlsFindRC searches a string in reverse order for a specified single-byte or multibyte character. It then returns the offset, from the beginning of the string, at which the character is found.

EnlsFindRC can be used with the EnlsDeleteChar operation. After the application has found the location of a specific character with EnlsFindRC, it can then call EnlsDeleteChar to delete the character. (See EnlsDeleteChar.)

Note that EnlsFindRC should be used in place of high-level language procedures that search in reverse order for values within a string. For example, an application should use EnlsFindRC instead of the PL/M FindRB procedure.

EnlsFindRC differs from EnlsFindC in that the former searches in reverse order for a character. (See EnlsFindC.)

The value 65535 is returned at *pwOffsetRet* if the specified character is not found. Status code 15208 ("Bad string size") is returned if the specified string size is a negative value.

Procedural Interface

EnlsFindRC (pbString, pChar, cbString, pwOffsetRet): ercType

where

pbString

is the memory address of the character string to be searched.

pChar

is the memory address of the character to be located within the character string. This character may be one to four bytes in size.

cbString

is the length of the character string (in bytes) to be searched.

pwOffsetRet

is the memory address of a word where the character's offset is returned.

Request Block

EnlsFindRC is an object module procedure in Enls.lib.

EnlsGetChar (pb, cb, pCharRet, pcbCharRet, bType): ercType

NOTE: An application should use EnlsGetChar and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsGetChar retrieves a single-byte or multibyte character from either the keyboard or an application-supplied memory area. It is similar in function to ReadKbdInfo, but offers the additional option of reading from a memory area. (See ReadKbdInfo.)

Existing applications should use EnlsGetChar in place of ReadKbd or ReadKbdDirect. (See ReadKbd and ReadKbdDirect.) Unlike the latter operations, EnlsGetChar can retrieve both single-byte and multibyte characters.

Status code 15205 ("End of string") is returned if the string at *pb* is too small to hold a character.

For details on Extended Native Language Support, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsGetChar (pb, cb, pCharRet, pcbCharRet, bType): ercType

where

pb
cb

describe the character string from which to retrieve a character. If *pb* is 0, EnlsGetChar retrieves a character from the keyboard.

pCharRet

is the memory address of a double word where the character is to be returned.

pcbCharRet

is the memory address of a word where the size (in bytes) of the character will be returned.

bMode

is the type of input stream from which to read a character. Each value indicates the following:

Value	Description
0	Return the character by calling ReadKbdDirect with mode set to 0. EnlsGetChar ignores the *pb* and *cb* parameters.
1	Return the character by calling ReadKbdDirect with mode set to 1. EnlsGetChar ignores the *pb* and *cb* parameters.
2	Return the character by calling ReadKbdDirect with mode set to 2. EnlsGetChar ignores the *pb* and *cb* parameters.

Value	Description
3	Return the character by calling ReadKbdDirect with mode set to 3. EnlsGetChar ignores the *pb* and *cb* parameters.
128	Return the character by calling ReadKbd. EnlsGetChar ignores the *pb* and *cb* parameters.
130	Return the character from a caller-supplied memory area specified by the *pb* and *cb* parameters.

Request Block

EnlsGetChar is an object module procedure in Enls.lib.

This page intentionally left blank

EnlsGetCharWidth (pChar, pwWidthRet): ercType

NOTE: An application should use EnlsGetCharWidth and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsGetCharWidth returns the size, in bytes, of a specified character code.

EnlsGetCharWidth can be used in conjunction with the PutFrameChars operation. (See PutFrameChars.) When the application calls PutFrameChars, it must specify the size, in bytes, of the character that will be written to the screen. EnlsGetCharWidth provides a character width that can be passed on to a subsequent PutFrameChars operation.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsGetCharWidth (pChar, pwWidthRet): ercType

where

pChar

is the memory address of the character whose length is to be returned.

pwWidthRet

is the memory address of a word where the length of the character code is to be returned.

Request Block

EnlsGetCharWidth is an object module procedure in Enls.lib.

EnlsGetPrevChar (pbString, cbString, oChar, pCharRet, pcbCharRet):
ercType

NOTE: An application should use EnlsGetPrevChar and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsGetPrevChar returns the single-byte or multibyte character whose position precedes the specified offset within a string. EnlsGetPrevChar also returns the length of the returned character.

Status code 15200 ("Bad offset") is returned if *oChar* does not point to the first byte of a multibyte character code. This status code is also returned if no characters precede the specified string offset. Status code 15205 ("End of string") is returned if the *cbString* parameter is set to 0.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsGetPrevChar (pbString, cbString, oChar, pCharRet, pcbCharRet):
ercType

where

pbString
cbString

describe the character string containing the character to be returned.

oChar

is the string offset. EnlsGetPrevChar returns the character positioned before this offset.

pCharRet

is the memory address of a double word where the character is to be returned.

pcbCharRet

is the memory address of a word where the size (in bytes) of the character is to be returned.

Request Block

EnlsGetPrevChar is an object module procedure in Enls.lib.

EnlsInsertChar (pbString, cbString, oChar, sBuffer, pChar, fTypeOver, pcbStringRet): ercType

NOTE: An application should use EnlsInsertChar and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code. For details on Extended Native Language Support, see "Native Language Support" in the CTOS Operating System Concepts Manual.

Description

EnlsInsertChar inserts a single-byte or multibyte character into a character string. If the application needs to preserve underlying text, EnlsInsertChar creates space for the inserted character by moving other characters in the string to the right.

When EnlsInsertChar overwrites a multibyte character with a single-byte character, all bytes of the multibyte character are deleted. Conversely, if EnlsInsertChar overwrites a single-byte character with a multi-byte character, it creates space for the multibyte character by shifting other characters to the right.

EnlsInsertChar is often used to modify text strings that appear on a video display. For example, the application can call QueryFrameChar to obtain a copy of the string. After it has used EnlsInsertChar, the application can then call PutFrameChars to display the modified string. (See PutFrameChars and QueryFrameChar.)

Status code 15200 ("Bad offset") is returned if the *oChar* parameter points to the second byte of a multibyte character code. This status code is also returned if the specified string offset is greater than the size of the string. Status code 15202 ("Buffer full") is returned if the buffer is not large enough to hold the resulting string.

Procedural Interface

EnlsInsertChar (pbString, cbString, oChar, sBuffer, pChar, fTypeOver, pcbStringRet): ercType

where

pbString
cbString

describe the character string into which the character is to be inserted.

oChar

is the offset, from the beginning of the string, where the character is to be inserted.

sBuffer

is the size (in bytes) of the buffer in which the string is located.

pChar

is the memory address of the character to be inserted. This character may be one to four bytes in size.

fTypeOver

is a one-byte flag which, when set to TRUE, causes the character at the specified string offset to be overwritten by the inserted character.

pcbStringRet

is the memory address of a word where the length of the resulting character string is returned.

Request Block

EnlsInsertChar is an object module procedure in Enls.lib.

EnlsMapCharToStdValue (pChar, pbStdValue): ercType

NOTE: An application should use EnlsMapCharToStdValue and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsMapCharToStdValue maps a character code from the currently loaded character set to a character code from the standard character set. The application provides the memory address of the character code for which an equivalent character is provided. (See Appendix B for a description of the standard character set.)

Status code 15207 ("No equivalent character") is returned if an equivalent standard character code does not exist.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsMapCharToStdValue (pChar, pbStdValue): ercType

where

pChar

is the memory address of the character code (1 to 4 bytes) from the currently loaded character set.

pbStdValue

is the memory address where the character code from the standard character set is returned.

Request Block

EnlsMapCharToStdValue is an object module procedure in Enls.lib.

EnlsMapStdValueToChar (bStdValue, pCharRet, pcbCharRet): ercType

NOTE: An application should use EnlsMapStdValueToChar and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsMapStdValueToChar maps a character code from the standard character set to an equivalent character code that is based on the currently loaded character set. (See Appendix B for a description of the standard character set.)

Status code 15207 ("No equivalent character") is returned if an equivalent character code does not exist.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsMapStdValueToChar (bStdValue, pCharRet, pcbCharRet): ercType

where

bStdValue

is a one-byte character code from the standard character set.

pCharRet

is the memory address where the equivalent character code from the currently loaded character set is to be returned.

pcbCharRet

is the memory address of a word to which the size (in bytes) of the equivalent multibyte character code is returned.

Request Block

EnlsMapStdValueToChar is an object module procedure in Enls.lib.

EnlsQueryBoxSize (nCols, nLines, pwBoxSizeRet): ercType

NOTE: An application should use EnlsQueryBoxSize and other ENLS calls over non-ENLS equivalents. The ENLS operations allow applications to process single-byte and multibyte characters without changes to the program code.

Description

EnlsQueryBoxSize calculates the number of bytes required to save the contents of a box. The calling program specifies the screen coordinates of the box.

EnlsQueryBoxSize is especially useful in Asian language environments, where each character may consist of multiple bytes. In this case, the number of bytes to be saved cannot be solely determined by the number of rows and columns that a box occupies.

EnlsQueryBoxSize can be used in conjunction with QueryFrameCharsAndAttrs. Before an application saves a portion of the screen, it can use EnlsQueryBoxSize to determine the amount of memory required for the screen data. The application then can call QueryFrameCharsAndAttrs with the value returned by EnlsQueryBoxSize.

For details on Extended Native Language Support, see the *CTOS Operating System Concepts Manual.*

Procedural Interface

EnlsQueryBoxSize (nCols, nLines, pwBoxSizeRet): ercType

where

nCols

is the number of columns occupied by the box.

nLines

is the number of lines (rows) occupied by the box.

pwBoxSizeRet

is the memory address of a word to which the number of bytes required to save the contents of a box is returned.

Request Block

EnlsQueryBoxSize is an object module procedure in Enls.lib.

ErrorExit (ercTermination)

Description

ErrorExit terminates the current application program and passes a specified status code to the exit run file. ErrorExit never returns to the calling process.

ErrorExit is exactly like the Exit operation except that the status code in ErrorExit is explicit. Exit's status code is implicitly 0. Both operations place the specified status code in the Application System Control Block of an application partition.

ErrorExit performs the following functions:

1. Terminates the current application program.
2. Disconnects interrupt handlers of the partition and terminates all processes of the partition.
3. Terminates keyboard (application partition only), timer, and communications requests, and waits until all disk and printer input/output activity has ceased.
4. In an application partition only, resets the keyboard to character mode. Discards the content of the type-ahead buffer if the keyboard was in unencoded mode and/or the status code is nonzero.
5. In an application partition only, re-enables the ACTION-FINISH feature and discards the action code (if any).
6. In an application partition only, closes the Submit or recording file if the status code is nonzero.
7. Closes all files opened for the application partition except those marked long-lived by the OpenFileLL or SetFhLongevity operations. (See "File Management" in the *CTOS Operating System Concepts Manual* for more information.)

8. Releases for reuse all partition memory that was allocated as short-lived.

9. Verifies that the exit run file exists, that it contains a valid arrangement of segments, and that it fits in the application partition memory (the operating system terminates if this verification fails).

10. Allocates a short-lived memory segment large enough to contain the exit run file. If sufficient application partition memory to load the exit run file cannot otherwise be allocated, then long-lived memory is reset (that is, released to the common pool of unallocated memory) before the exit run file is loaded. If an exit run file does not exist or is not specified, the partition goes vacant.

11. Reads the run file into the partition.

12. Relocates all intersegment references to accommodate the memory address at which the run file is loaded.

13. Creates a process to be scheduled at the default priority. The initial values loaded into the segment registers (Code Segment (CS), Data Segment (DS), Stack Segment (SS), Extra Segment (ES)), the Stack Pointer (SP), and the Instruction Pointer (IP) are derived from information in the run file header.

ErrorExit has no effect on the allocation of long-lived memory except as noted in step 10 above. If necessary, the exit run file is left open to accommodate code swapping of the exit run file. The file handle of the open run file is placed in the Application System Control Block of the partition.

Procedural Interface

ErrorExit (ercTermination)

where

ercTermination

is a 16-bit status code to be placed in the Application System Control Block of the application partition for examination by the exit run file. In an application partition only, a nonzero status code causes the content of the type-ahead buffer to be discarded and the Submit or recording file to be closed.

Request Block

ErrorExit is a system-common procedure.

This page intentionally left blank

ErrorExitString (erc, pbString, cbString)

Description

ErrorExitString returns a string to the exit run file. It is typically used to return a string to the Executive to be displayed to the video device along with a subsystem error message.

Procedural Interface

ErrorExitString (erc, pbString, cbString)

where

erc

is a status code to be used as a parameter to a call to ErrorExit.

pbString
cbString

describe a string to be placed in the Application System Control Block of the application partition for examination by the exit run file.

Request Block

ErrorExitString is an object module procedure.

This page intentionally left blank

EstablishQueueServer (pbQueueName, cbQueueName, queueType, fUniqueServer): ercType

Description

EstablishQueueServer is issued by the caller to notify the Queue Manager that the caller intends to serve the specified queue. The caller must issue EstablishQueueServer before using any of the queue server operations (such as MarkKeyedQueueEntry or UnmarkQueueEntry) to process queue entries in the queue.

Procedural Interface

EstablishQueueServer (pbQueueName, cbQueueName, queueType, fUniqueServer): ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to the queue name specified when the queue was installed.

queueType

is the type of the queue (an integer less than or equal to 255), which is used in a consistency check. The Queue Manager checks the type against the type when the queue was installed.

fUniqueServer

is a flag determining whether or not the requesting program is to be the unique server for the specified queue.

TRUE means the requesting program intends to become the unique queue server for the specified queue. If the queue already is being served, the Queue Manager returns status code 914 ("Queue already served"). Otherwise the operation succeeds, and other programs are prevented from becoming queue servers for the queue.

FALSE means the caller does not intend to become the unique queue server for the specified queue.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	146
12	queueType	2	
14	fUniqueServer	1	
15	reserved	1	
16	pbQueueName	4	
20	cbQueueName	2	

Exit

Description

Exit terminates the current application program and passes a normal status code to the exit run file. Exit never returns to the calling process.

Exit is exactly like the ErrorExit operation except that the status code in Exit is implicit (see the ErrorExit operation). That is,

```
Exit();
```

is equivalent to

```
ErrorExit (0);
```

Exit places a normal successful status code (0) in the Application System Control Block of an application partition.

Procedural Interface

Exit

Request Block

Exit is a system-common procedure.

This page intentionally left blank

ExitAndRemove

NOTE: This operation does not return a status code.

Description

ExitAndRemove terminates the current application program and removes the specified vacant partition.

The user number is deallocated and becomes available to be reissued in response to a call to CreatePartition.

ExitAndRemove performs the same functions as the combined functions of VacatePartition and RemovePartition. (See VacatePartition and RemovePartition.) It is useful for freeing the user number of a system service that has deinstalled.

Procedural Interface

ExitAndRemove

Request Block

ExitAndRemove is a system common procedure.

This page intentionally left blank

ExitListQuery (lh, pProcInfoRet, sProcInfoRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

ExitListQuery returns a block of exit procedure information for the specified dynamic link library (DLL) in the buffer provided by the caller. The exit procedures were previously registered by a call to ExitListSet. (See ExitListSet.) If the buffer is too small, the data returned is truncated.

Procedural Interface

ExitListQuery (lh, pProcInfoRet, sProcInfoRet): ercType

where

lh

is the DLL handle (word).

pProcInfoRet

is the memory address of the buffer to which the block of exit procedure information is written. The block has the following format:

Offset	Field	Size (Bytes)	Description
0	cProc	2	Number of exit procedures
2	rgpProc (*) pointer	cProc*4	Array of memory addresses of exit procedures.

sProcInfoRet

is the size (word) of the information block buffer.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	471
12	lh	2	
14	reserved	4	
18	pProcInfoRet	4	
22	sProcInfoRet	2	

ExitListSet (wFlags, pProc): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Caution: *The procedures operated on by ExitListSet must be large model: no DS can be assumed to be in effect when such procedures are called.*

Description

ExitListSet updates the list of procedures to be run when the current client terminates. The operation can add a procedure to the list or remove one. The list of exit procedures is short-lived. It is reset after the client terminates.

Procedural Interface

ExitListSet (wFlags, pProc): ercType

where

wFlags

is a word value. The bits have the following meanings:

Bit(s)	Meaning
0	If the value of this bit is 0, remove the procedure. If the value is 1, add the procedure.
1-7	Are reserved.

8-15 Contain the priority value of the procedure in the exit list queue. Lower priority procedures are dispatched before higher priority procedures. The priority ranges and their assignments are

Range	Assignment
0 to 1Fh	Reserved for operating system.
20h to 9Fh	Available for general use.
A0h to BFh	Reserved for Presentation Manager DLLs.
C0h to FFh	Reserved for Presentation Manager (DOS emulation).

pProc

is the memory address of the exit procedure to be added or removed.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	429
12	wFlags	2	
14	pProc	4	

This page intentionally left blank.

ExpandAreaLL (cBytes, sa, pRaRet): ercType

Description

ExpandAreaLL allocates additional memory of the specified size within the specified long-lived segment.

Programs designed to execute in both real and protected modes should use the ExpandAreaLL operation to allocate additional memory in a long-lived segment.

Procedural Interface

ExpandAreaLL (cBytes, sa, pRaRet): ercType

where

cBytes

is the amount in bytes by which the segment is to be expanded. Status code 400 ("Memory not available") is returned if the resulting segment would be larger than 64K bytes.

sa

is the segment address (high-order two bytes of a memory address) of the segment to be expanded.

pRaRet

is the memory address of a word to which the offset (low-order two bytes of a memory address) of the newly allocated memory is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	280
12	cBytes	2	
14	sa	2	
16	reserved	2	
18	pRaRet	4	
22	sRaRetMax	2	2

ExpandAreaSL (cBytes, sa, pRaRet): ercType

Description

ExpandAreaSL allocates additional memory of the specified size within the specified short-lived segment. The specified segment must have been created by a prior call to AllocAreaSL (except when using the Linker's DS Allocation facility and ExpandAreaSL to expand the static data segment, DGroup). The segment grows by expanding downward (toward lower offsets).

Programs designed to execute in both real and protected modes should use ExpandAreaSL to allocate additional memory in a short-lived segment.

Procedural Interface

ExpandAreaSL (cBytes, sa, pRaRet): ercType

where

cBytes

is the amount in bytes by which the segment is to be expanded. Status code 400 ("Memory not available") is returned if the resulting segment would be larger than 64K bytes.

sa

is the segment address (high-order two bytes of a memory address) of the segment to be expanded.

pRaRet

is the memory address of a word to which the offset (low-order two bytes of a memory address) of the newly allocated memory is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	279
12	cBytes	2	
14	sa	2	
16	reserved	2	
18	pRaRet	4	
22	sRaRetMax	2	2

ExpandDateTime (dateTime, pExpDateTimeRet): ercType

Description

ExpandDateTime converts the date and time from the System Date/Time format to the Expanded Date/Time format in which year, month, day of month, and so forth are represented as discrete fields.

(For details on the expanded format, see "Expanded Date/Time Format" in Chapter 4, System Structures." The System Date/Time format is also described in "Utility Operations" in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

ExpandDateTime (dateTime, pExpDateTimeRet): ercType

where

dateTime

is the 32-bit date/time in the System Date/Time format.

pExpDateTimeRet

is the memory address of an 8-byte block where the date/time in the Expanded Date/Time format is to be returned.

Request Block

ExpandDateTime is an object module procedure.

This page intentionally left blank

ExpandLocalMsg (pRgSd, sRgSd, pbMsg, cbMsg, pBufRet, sBufRet, pCbRet, fBs): ercType

Description

ExpandLocalMsg expands any macro definitions contained in a message that resides in local memory.

ExpandLocalMsg generally is used in conjunction with GetMsgUnexpanded which copies a message from a message file on disk to an area in local memory.

A message macro consists of number(s) and/or letter(s) preceded by a percent sign (%). (See Appendix F for a description of the message file macros.)

For example, given a message of the form:

Copying %0 to %1 ...

if you supply two strings (such as foo and mumble) in the array of string descriptors described by pRgSd and sRgSd, ExpandLocalMsg would expand the message to:

Copying foo to mumble ...

Procedural Interface

ExpandLocalMsg (pRgSd, sRgSd, pbMsg, cbMsg, pBufRet, sBufRet, pCbRet, fBs): ercType

where

pRgSd
sRgSd

describe an array of string descriptors. Each element of the array describes a string that is inserted in the message in place of a macro. (See Appendix F for details.)

String descriptors are 6-byte blocks of memory where the first 4 bytes contain the memory address of the string and the last 2 bytes contain the size of the string.

pbMsg
cbMsg

describe the message (with internal macros).

pBufRet

is either the memory address of a buffer to which the expanded message is placed; or, if fBs (described below) is TRUE, the address of a byte stream work area (BSWA). The BSWA describes a byte stream (for example, bsVid) to which the expanded message is output.

sBufRet

is the size of the buffer.

pcbRet

is the memory address of a word to which the actual length of the expanded message is returned.

fBs

TRUE implies that pBufRet is the address of a BSWA. FALSE implies that pBufRet is the address of a buffer.

Request Block

ExpandLocalMsg is an object module procedure.

This page intentionally left blank

ExpandSpec (pbFileSpec, cbFileSpec, pbPassword, cbPassword, pSpecRet, sSpecMax, specType): ercType

Description

ExpandSpec expands an incomplete file specification to a full specification, which contains the node, volume, directory, file, and password, if required, of the caller. ExpandSpec accepts the incomplete specification values from pbFilespec and pbPassword. It completes the specification by obtaining information from the caller's default path.

ExpandSpec can only expand specifications for users at the workstation where the request was issued. The returned specification is in canonical form. For example, the volume name field returned will always contain the name assigned to the volume when it was initialized rather than [Sys] or [Scr]. For unmounted or uninitialized volumes, however, the volume name field will be returned as the device name, such as [d0].

In the case of the <$> directories, the expanded directory name will always be returned.

Procedural Interface

ExpandSpec (pbFileSpec, cbFileSpec, pbPassword, cbPassword, pSpecRet, sSpecMax, specType): ercType

where

pbFileSpec
cbFileSpec

describe the partial specification to be expanded.

pbPassword
cbPassword

describe the password, if any.

pSpecRet
sSpecMax

describe the expanded file specification format returned.

specType

indicates whether the specification is a volume, directory, or file type, as follows:

value	Meaning
1	volume
2	directory
3	file

For example, Files.run (which is used by the Executive **Files** command) specifies Type 3 because the program expects a file name in the specification to be expanded.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	253
12	specType	2	
14	reserved	4	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	
32	pSpecRet	4	
36	sSpecMax	2	

This page intentionally left blank

FatalError (ercTermination)

Description

FatalError terminates operation of the application program and passes a specified status code to the exit run file. The standard version of FatalError consists of a call to the ErrorExit operation.

Standard object module procedures call FatalError rather than the ErrorExit operation when they encounter inconsistencies. This allows the system builder to easily substitute a user-written version of FatalError that does one of the following, for example:

- calls the Crash operation because it causes a crash dump to be written that is useful in debugging
- calls the Crash operation because the application program wants to crash the system when it malfunctions. (Note that Crash should only be used in this way in those rare cases when the error detected might otherwise affect other programs and is one from which the program cannot recover.)
- provides special logic to attempt an orderly system shutdown when the application program detects a malfunction. Such code is best included in a user-written version of FatalError
- invokes the Debugger

System services should use Crash instead of FatalError to terminate execution. This avoids hanging the system in a way that would be difficult to trace.

Procedural Interface

FatalError (ercTermination)

where

ercTermination

is a 16-bit status code to be placed in the Application System Control Block for interrogation by the exit run file. A nonzero status code causes the content of the type-ahead buffer to be discarded and the submit or recording file to be closed.

Request Block

FatalError is an object module procedure.

FComparePointer(p1, p2, bCompareMode): flagType

Description

FComparePointer returns TRUE if p1=p2. The bCompareMode parameter defines the meaning of pointer (logical address) equality. The codes and descriptions for bCompareMode are as follows:

Code	Description
1	Return TRUE if p1 and p2 reference the same byte location in the linear address space. For example, in real mode the logical addresses 10:8 and 0:108 are equal. In protected mode, the logical addresses 88:4 and 9C:2A0 are equal only if the linear base address stored in the Global Descriptor Table (GDT) segment descriptor identified by selector 88h, plus 4, equals the linear base address stored in the Local Descriptor Table (LDT) descriptor identified by selector 9Ch, plus 2A0h. This form of comparison is more expensive and should be used only when the possibility of alias addresses exists.
2	If in real mode, return TRUE only if p1 and p2 have the same 32-bit binary value. In protected mode, return TRUE if p1 and p2 have the same binary value except for possible differences in the RPL field of the selectors. This is the typical meaning of pointer equality.

Procedural Interface

FComparePointer(p1, p2, bCompareMode): flagType

where

p1

is the first logical address (pointer) for comparison.

p2

is the second logical address (pointer) for comparison.

bCompareMode

determines how the test for equality is made.

Request Block

FComparePointer is an object module procedure.

FieldEdit (iFrame, iCol, iLine, bFieldType, cbFieldLength, pbText, cbText, sTextMax, pcbRet, pChRet): ercType

NOTE: To ensure portability across different language environments, an application should use EnlsFieldEdit instead of FieldEdit. (See EnlsFieldEdit.)

Description

FieldEdit displays and edits a line of text on the video display. As the user types characters, FieldEdit updates a text descriptor using the TextEdit operation and updates the display to show the current state of the line. This operation returns when the user types a character that TextEdit considers invalid. (See TextEdit.)

FieldEdit returns status code 500 ("Bad frame address") if the character buffer specified is smaller than the length of the display field.

Procedural Interface

FieldEdit (iFrame, iCol, iLine, bFieldType, cbFieldLength, pbText, cbText, sTextMax, pcbRet, pChRet): ercType

where

iFrame

specifies the frame (word).

iCol

specifies the horizontal coordinate (word) within *iFrame* at which the first character of the field is to be displayed.

iLine

specifies the vertical coordinate (word) within *iFrame* at which the first character of the field is to be displayed.

bFieldType

is a byte value that determines how the user-entered text is displayed. The values of *bFieldType* and their corresponding meanings are

Value	Meaning
0	display the characters the user types (normal display)
1	display a pound sign character (#) for each character the user types (password display)

cbFieldLength

is a word value that specifies the width (in characters) of the display field.

pbText
cbText

describe the initial text string to be edited.

sTextMax

is the maximum size (word) of the returned text string.

pcbRet

is the memory address of the word where the actual length of the returned text string is to be written.

pChRet

is the memory address of the byte where the exit character is to be written.

Request Block

FieldEdit is an object module procedure.

This page intentionally left blank

FillBufferAsyncC (pBs, cbMin, cbMax, pPbRet, pCbRet, idExch, pMsg, rqCode): ercType

Description

FillBufferAsyncC is the asynchronous form of the FillBufferC. (See FillBufferC.)

FillBufferAsyncC differs from FillBufferC as follows:

- Both a cbMin and a cbMax can be specified. The cbMin is only a hint, however; FillBufferAsyncC waits for at least cbMin chars to be available unless forced to return earlier with fewer characters. In addition to various error conditions, earlier return can occur even within an ercOK situation, such as when the circular receive queue wraps around. The caller cannot assume he will get at least cbMin characters; he must always check cbRet. (Provided that cbMin is nonzero, cbRet is always at least 1 on an ercOK return.)

- It is possible to prevent all forms of blocking by specifying cbMin = 0. This is similar in effect to specifying a receive timeout of 0, except that the status code is 0 ("ercOK") rather than status code 602 ("No character available") when cbRet = 0 is returned.

- It is also possible to make an asynchronous call, which never actually blocks, but returns status code 2347 ("Cannot complete now") if it would have blocked. When the data is available, the caller is notified by a message.

This feature is controlled by the idExch, pMsg, and rqCode parameters.

To make an asynchronous call, supply the ID of an exchange, a message (or a unique address, such as 0), and optionally a request code (see below); then check for status code 2347 ("Cannot complete now") after the call.

If "Cannot complete now" is returned, this indicates that a message will arrive at the exchange later. The caller should expect a message from the interrupt service routine when the remainder of the data is available, or a timeout message from the operating system (whichever occurs first; never both). A process (not necessarily the same process that got the "Cannot complete now") must wait (or poll with Check) at the exchange for the message.

When "Cannot complete now" is returned, cbRet will be 0. Even if part of the cbMin was available, no data will have been returned to the caller. (Use a smaller cbMin, such as 0 or 1, to get that effect.) To get the data when it has arrived, the caller must call again after receiving the message; if the same parameters are used, the call will succeed this time. It is illegal to call again until the message (or a timeout message) is sent; calls prior to this time will be returned with status code 2348 ("Wrong time to call"). (An exception to this rule is that the receive-type and transmit-type operations are completely independent, and never affect each other. For example, after status code 2347 ("Cannot complete now") is returned by FillBufferAsyncC, it is still valid to call FlushBufferAsyncC but not FillBufferAsyncC again until the message arrives.)

Status code 2347 ("Cannot complete now") can occur only when the following conditions exist:

- cbMin > 0
- there are fewer than cbMin chars in the receive queue
- the caller specified a nonzero idExch
- the caller did not specify an immediate timeout (0 seconds)

- The optional rqCode parameter is included only for completeness. It is placed in the rqCode field of the Timer Request Block (TRB) (timeout message), which the caller receives in case of a timeout. (See Chapter 4, "System Structures," for the format of the TRB.) The rqCode parameter has no meaning except to the caller. The caller could use it (or memory address comparison) to distinguish the timeout message from his own completion message, for example. In particular, it does not serve to route the timeout message, which is always sent via PSend to the caller's idExch.

- The ercRet field in the TRB is set to status code 602 ("No character available"). The caller can also use this field to distinguish the TRB from its own message.

- FillBufferC is equivalent to FillBufferAsyncC with cbMin = 1 and idExch = 0.

Procedural Interface

FillBufferAsyncC (pBs, cbMin, cbMax, pPbRet, pCbRet, idExch, pMsg, rqCode): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

cbMin

is the minimum number of characters that the caller prefers (word). This is only a hint; fewer characters may be returned in some cases. If cbMin = 0, all forms of blocking (waiting) are prevented.

cbMax

is the maximum number of characters that the caller can accept (word).

pPbRet

is the memory address to which the address of the data is to be returned.

pCbRet

is the memory address to which the number of characters actually received is to be returned.

idExch pMsg rqCode

are used to make an asynchronous (non-blocking) call, and are described above. For a synchronous call, including a call in which all forms of blocking are prevented (cbMin = 0), these parameters should be zero.

Request Block

FillBufferAsyncC is an object module procedure.

FillBufferC (pBs, cbMax, pPbRet, pCbRet): ercType

Description

FillBufferC is used to read characters from the receive queue that have been received at the serial port.

(See "Communications Programming" in the *CTOS Operating System Concepts Manual* for more information.)

There is also an asynchronous form of this call (See FillBufferAsyncC.)

Procedural Interface

FillBufferC (pBs, cbMax, pPbRet, pCbRet): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

cbMax

is the maximum number of characters that the caller can accept (word).

pPbRet

is the memory address to which the address of the data is to be returned.

pCbRet

is the memory address to which the number of characters actually received is to be returned.

Request Block

FillBufferC is an object module procedure.

This page intentionally left blank

FillFrame (iFrame, iCol, iLine, bToFill, cToFill): ercType

NOTE: To draw a line or a box, an application should use EnlsDrawLine or EnlsDrawBox in place of FillFrame. EnlsDrawLine, EnlsDrawBox, and other ENLS operations allow an application to run in different language environments without changes to the program code.

Description

FillFrame is similar to PutFrameAttrs except FillFrame fills a specified frame with the same character (rather than the same character attribute) for a range of character positions within the frame. The character is moved into the frame first left to right and then top to bottom.

Procedural Interface

FillFrame (iFrame, iCol, iLine, bToFill, cToFill): ercType

where

iFrame

specifies the index of the frame descriptor in the Video Control Block (VCB). An index value of 255 is treated as a special case and is reserved for internal use only.

iCol
iLine

specify the horizontal and vertical coordinates within the frame at which to begin filling with the character.

btoFill

specifies the character used to fill the frame.

ctoFill

specifies the number of bytes to be filled with the specified character.

Request Block

FillFrame is a system common procedure.

FillFrameRectangle (iFrame, iCol, iLine, cbCols, cbLines, bChar, bAttr): ercType

NOTES:

1. *This operation is supported on protected mode operating systems only.*
2. *When using standard VGA on EISA/ISA-bus workstations, this operation may display some attributes (specifically, bold, underlining, struckthrough, and blinking) differently than expected. System configuration options allow you to substitute color for these attributes. See the* CTOS System Administration Guide *for more information.*

Description

FillFrameRectangle fills a specified rectangle within a specified frame with characters and corresponding attributes. The characters and attributes are moved into the rectangle first left to right and then top to bottom.

If *cbLines* = 0, however, *cbCols* is the total number of characters that will be used to fill the frame (rather than a rectangle within it) starting at the horizontal and vertical coordinates, *iCol* and *iLine*. Characters in excess of one frame line wrap around to the next line.

Procedural Interface

FillFrameRectangle (iFrame, iCol, iLine, cbCols, cbLines, bChar, bAttr): ercType

where

iFrame

specifies the index of the frame descriptor in the Video Control Block (VCB). An index value of 255 is treated as a special case and is reserved for internal use only.

iCol
iLine

specify the horizontal and vertical coordinates within the frame at which to begin filling with the character.

cbCols

specifies the width (number of columns to be filled) of the rectangle.

cbLines

specifies the height (number of lines to be filled) of the rectangle. If *cbLines* = 0, *cbCols* is the total number of characters that will be used to fill the frame starting at the horizontal and vertical coordinates, *iCol* and *iLine*.

bChar

is the character (byte) to be used as the fill character.

bAttr

is the corresponding attribute.

Request Block

FillFrameRectangle is a system common procedure.

FillInRectangle (iFrame, iCol, iLine, cbCols, cbLines, bChar, bAttr): ercType

NOTES:

1. *This operation is similar to FillFrameRectangle except that it is a system library procedure that works on all operating system versions. This operation is for use by programs that need to execute on real mode operating systems.*
2. *When using standard VGA on EISA/ISA-bus workstations, this operation may display some attributes (specifically, bold, underlining, struckthrough, and blinking) differently than expected. System configuration options allow you to substitute color for these attributes. See the* CTOS System Administration Guide *for more information.*

Description

FillInRectangle fills a specified rectangle within a specified frame with characters and corresponding attributes. The characters and attributes are moved into the rectangle first left to right and then top to bottom.

If *cbLines* = 0, however, *cbCols* is the total number of characters that will be used to fill the frame (rather than a rectangle within it) starting at the horizontal and vertical coordinates, *iCol* and *iLine*. Characters in excess of one frame line wrap around to the next line.

Procedural Interface

FillInRectangle (iFrame, iCol, iLine, cbCols, cbLines, char, attr): ercType

where

iFrame

specifies the index of the frame descriptor in the Video Control Block (VCB). An index value of 255 is treated as a special case and is reserved for internal use only.

iCol
iLine

specify the horizontal and vertical coordinates within the frame at which to begin filling with the character.

cbCols

specifies the width (number of columns to be filled) of the rectangle.

cbLines

specifies the height (number of lines to be filled) of the rectangle. If *cbLines* = 0, *cbCols* is the total number of characters that will be used to fill the frame starting at the horizontal and vertical coordinates, *iCol* and *iLine*.

char

is the character to be used as the fill character.

attr

is the corresponding attribute.

Request Block

FillInRectangle is an object module procedure.

FilterDebugInterrupts (sgTss, exchNotify, fV86Only): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

Description

FilterDebugInterrupts directs the Debugger to pass through Debugger interrupts (single step, breakpoint) on a per process basis by sending messages to an exchange.

Procedural Interface

FilterDebugInterrupts (sgTss, exchNotify, fV86Only): ercType

where

sgTss

is the Task State Segment (TSS) selector that identifies the process for which Debugger interrupts are filtered.

exchNotify

is the exchange to which the Debugger will send a message when a Debugger interrupt occurs within the process identified by *sgTss*. The lower two bits of the message contain the interrupt type, and the upper fourteen bits contain the TSS selector.

fV86Only

if TRUE, identifies *sgTss* as a process that executes using the virtual 8086 mode provided by the 80386 microprocessor.

(For details on TSS and virtual 8086 mode, see the *80386 Programmer's Reference Manual.)*

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	335
12	sgTss	2	
14	exchNotify	2	
16	fV86Only	1	

FlushBufferAsyncC (pBs, pb, cbMin, cbMax, pCbRet, idExch, pMsg, rqCode): ercType

Description

FlushBufferAsyncC is the asynchronous form of the FlushBufferC operation. (See FlushBufferC.)

FlushBufferAsyncC differs from FlushBufferC as follows:

- To make an asynchronous call, supply the ID of an exchange and a message (or a unique address, such as 0); then check for status code 2347 ("Cannot complete now") after each call. This code is returned by the operation instead of waiting when *cbMin* cannot be satisfied immediately.

- "Cannot complete now" indicates that the caller's operation was not completed (because the transmit buffer is full). The operation would have waited (had *idExch* been 0), and the caller should now expect a message from the interrupt service routine (or a timeout message, whichever occurs first) when the remainder of the data can be queued for transmission.

- Note that if part of the caller's data is available to be queued, the data is queued immediately. *cbRet* indicates how much was queued. The message is sent after space for the remaining minimum data (*cbMin - cbRet*) is available.

- If status code 2347 ("Cannot complete now") is returned, the caller may not call again until the message (or a timeout message) is sent. Otherwise, status code 2348 ("Wrong time to call") is returned. (An exception to this rule is that the receive-type and transmit-type operations are completely independent, and never affect each other. For example, if FillBufferAsyncC returns status code 2347 ("Cannot complete now"), it is still valid to call FillBufferAsyncC but not FlushBufferAsyncC again until the message arrives.)

- After getting the message, it is the caller's responsibility to call again to queue the remaining characters.
- If the caller either does not want to wait or does not want the message notifying him of completion, *cbMin*=0 should be specified. In this case all forms of blocking are prevented and *idExch*, *pMsg*, and *rqCode* are not examined.
- The optional *rqCode* parameter is placed in the *rqCode* field of the Timer Request Block (TRB) (timeout message) that the user receives in case of a timeout. (See Chapter 4, "System Structures," for the format of the TRB.) The *rqCode* parameter has no meaning except to the user program, which might use it to distinguish the timeout message from the completion message. (It is not used to route the message.)
- FlushBufferC is equivalent to FlushBufferAsyncC with *cbMin* and *cbMax* equal to *cb*, and *idExch* equal to 0.

Procedural Interface

FlushBufferAsyncC (pBs, pb, cbMin, cbMax, pCbRet, idExch, pMsg, rqCode): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

pb

is the memory address of the data to be tramsmitted.

cbMin

is the minimum number of characters to queue (word).

cbMax

is the maximum number of characters to queue (word).

pCbRet

is the memory address to which the number of characters actually queued is to be returned.

idExch
pMsg
rqCode

are used to make an asynchronous (non-blocking) call, and are described above. For a synchronous call, including a call in which all forms of blocking are prevented (*cbMin*=0), these parameters should be 0.

Request Block

FlushBufferAsyncC is an object module procedure.

This page intentionally left blank

FlushBufferC (pBs, pb, cb, pCbRet): ercType

Description

FlushBufferC is used to write characters to the transmit queue, where they will be output to the RS-232 serial port.

There is also an asynchronous form of this call. (See FlushBufferAsyncC.)

(See "Communications Programming" in the *CTOS Operating System Concepts Manual.)*

Procedural Interface

FlushBufferC (pBs, pb, cb, pCbRet): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

pb
cb

describe the data to be output in characters (word). If *cb* is larger than the available space in the transmit queue, or larger than the entire transmit queue, FlushBufferC repeatedly fills the transmit queue and blocks (waits) for it to partially empty again, in order to complete the operation.

pCbRet

is the memory address to which the number of characters actually accepted for output is to be returned. *CbRet* is equal to *cb* except in the case of an error condition such as a transmit timeout.

Request Block

FlushBufferC is an object module procedure.

FMergedOs: flagType

Description

FMergedOs returns TRUE if the caller is running on CTOS II 3.3 or a later operating system version. Otherwise, FMergedOs returns FALSE. A TRUE value also indicates that the current version of GetModuleId provides X-Bus+ support.

Procedural Interface

FMergedOs: flagType

Request Block

FMergedOs is an object module procedure.

This page intentionally left blank

Format (fh, pBuffer, sBuffer, lfa, psDataRet): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

Description

Format initializes the surface of a floppy disk or other disk media to accommodate fixed-size data sectors. Format is used by the **Format Disk** command and is device-dependent. (See the *CTOS Executive Reference Manual* for details on the **Format Disk** command. Consult your hardware manuals for the control information required for a specific device.)

Procedural Interface

Format (fh, pBuffer, sBuffer, lfa, psDataRet): ercType

where

fh

is a file handle returned from an OpenFile operation that specifies a device.

pBuffer

is the memory address of the first byte of control information. The buffer must be word aligned. The size and contents of the control information depend upon the type of disk being formatted.

sBuffer

is the count of bytes of control information to be transferred. It must be a multiple of 2.

lfa

is the byte offset, from the beginning of the device, of the first sector to be initialized.

psDataRet

is the memory address of the word to which the count of bytes successfully transferred is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	38
12	fh	2	
14	lfa	4	
18	pBuffer	4	
22	sBuffer	2	
24	psDataRet	4	
28	ssDataRet	2	2

FormatDateTime (dateTime, psbTemplate, psbRet): ercType

Description

FormatDateTime converts the date and time from the System Date/Time format (described in "Utility Operations" in the *CTOS Operating System Concepts Manual*) to textual (string) format. The conversion is controlled by a template string, which contains embedded symbols that are replaced by the actual date and time values.

Programs should use NlsStdFormatDateTime or NlsFormatDateTime instead of FormatDateTime for ease in nationalization. (For details, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*) FormatDateTime is documented for historical reasons only.

Appendix E contains a description and an interpretation of the date and time templates. All the rules that apply to Native Language Support (NLS) templates also apply to FormatDateTime except that, in FormatDateTime, capitalization of the first letter of fields defined using the "n" or "w" characters is always forced. Thus these fields are always capitalized.

This rule has been retained for complete compatibility with earlier operating system versions.

Procedural Interface

FormatDateTime (dateTime, psbTemplate, psbRet): ercType

where

dateTime

is the 32-bit date/time in the System Date/Time format.

psbTemplate

is the address of the template string. The first byte of the template string is the length of the string; subsequent bytes contain the template itself. The template string is not modified by FormatDateTime.

psbRet

is the memory address to which the resultant string is to be returned. The first byte is the length of the returned string; subsequent bytes contain the characters of the string.

Request Block

FormatDateTime is an object module procedure.

FormatTime (psbDateTimeRet, pExpDateTimeStruct): ercType

Description

FormatTime converts the date and time in the Expanded Date/Time format (described in "Expanded Date/Time Format" in Chapter 4, "System Structures") into an ASCII string containing the day, date, and time, such as "Sun Oct 1, 1983 1:04 PM." This operation is equivalent to using NlsStdFormatDateTime with *iTemplate* equal to 5.

Programs should use NlsStdFormatDateTime instead of FormatTime for ease in nationalization. (For details, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

Appendix E contains a description and an interpretation of the date and time templates. All the rules that apply to Native Language Support (NLS) templates also apply to FormatTime except that, in FormatTime, capitalization of the first letter of fields defined using the "n" or "w" characters is always forced. Thus these fields are always capitalized.

Procedural Interface

FormatTime (psbDateTimeRet, pExpDateTimeStruct): ercType

where

psbDateTimeRet

is the memory address of a 31-byte buffer to which the string is copied. The first byte of the returned string is the actual size of the string.

pExpDateTimeStruct

is the memory address of the Expanded Date/Time format, as returned by the NlsParseTime, ParseTime, or ExpandDateTime operation.

Request Block

FormatTime is an object module procedure.

FormatTimeDt (psbDateRet, pExpDateTimeStruct): ercType

Description

FormatTimeDt converts the date and time in the Expanded Date/Time format (described in "Expanded Date/Time Format" in Chapter 4, "System Structures") into an ASCII string containing the date, such as "Oct 1, 1983." This operation is equivalent to using NlsStdFormatDateTime with *iTemplate* equal to 17 (11h).

Programs should use NlsStdFormatDateTime instead of FormatTimeDt for ease in nationalization. (For details, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

Appendix E contains a description and an interpretation of the date and time templates. All the rules that apply to Native Language Support (NLS) templates also apply to FormatTimeDt except that, in FormatTimeDt, capitalization of the first letter of fields defined using the "n" character is always forced. Thus this field is always capitalized.

Procedural Interface

FormatTimeDt (psbDateRet, pExpDateTimeStruct): ercType

where

psbDateRet

is the memory address of a 21-byte buffer to which the string is copied. The first byte of the returned string is the actual size of the string.

pExpDateTimeStruct

is the memory address of the Expanded Date/Time format, as returned by the NlsParseTime, ParseTime, or ExpandDateTime operation.

Request Block

FormatTimeDt is an object module procedure.

FormatTimeTm (psTimeRet, pExpDateTimeStruct): ercType

Description

FormatTimeTm converts the date and time in the Expanded Date/Time format (described in "Expanded Date/Time Format" in Chapter 4, "System Structures") into an ASCII string containing the time, such as "1:04 PM." It is equivalent to using NlsStdFormatDateTime with *iTemplate* equal to 9.

Programs should use NlsStdFormatDateTime instead of FormatTimeTm for ease in nationalization. (For details, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

Appendix E contains a description and an interpretation of the date and time templates. All the rules that apply to Native Language Support (NLS) templates also apply to FormatTimeTm except that, in FormatTimeTm, capitalization of the first letter of fields defined using the "a" character is always forced. Thus this field is always capitalized.

Procedural Interface

FormatTimeTm (psTimeRet, pExpDateTimeStruct): ercType

where

psTimeRet

is the 4-byte memory address of an 11-byte buffer into which the string is copied. The first byte of the returned string is the actual size of the string.

pExpDateTimeStruct

is the memory address of the Expanded Date/Time format, as returned by the NlsParseTime, ParseTime, or ExpandDateTime operation.

Request Block

FormatTimeTm is an object module procedure.

FormEdit (pNlsTableArea, iFrame, iCol, iLine, nFields, pbPrompt, cbPrompt, prgpFieldDescs, sMaxFields, maxCaptionLen, pChRet): ercType

Description

FormEdit displays a form and processes user responses to each field based on the descriptor for that field. User responses can be either of two types: the user can type in a string or select a choice from among a group of choices displayed. To make a selection, the user highlights the choice using the **Up Arrow**, **Down Arrow**, **Return**, and **Next** keys. If the Mouse Service is installed, a mouse may be used to highlight the items. To process a user-entered string response, FormEdit calls the FieldEdit operation.

Messages displayed by FormEdit can be located in a message file or in application memory. If a message file is used, the InitMsgFile operation must be called to open the file before this operation is used.

For each field, there may be (optional) online help. Help can be provided either in the form of a user-written procedure or in messages obtained from a message file. In the latter case, FormEdit calls the ZoomBox operation to create a box within which the help message and a prompt explaining how to exit the help box are displayed. FormEdit calls the UnzoomBox operation to collapse the help box when the user presses **Cancel** to exit.

Figure 3-1 and Figure 3-2 show two different examples of forms that could be displayed when FormEdit is called.

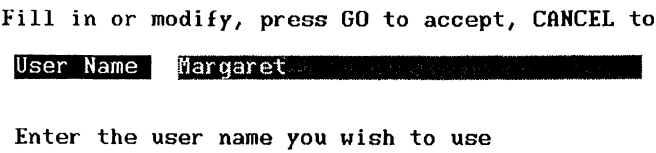

Figure 3-1. Form Showing a String Response

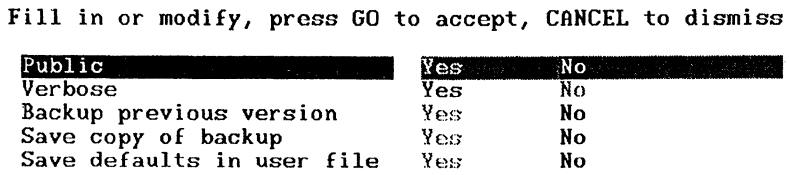

Figure 3-2. Form Showing Yes or No Choice Responses

In the example forms, the first line displayed is an (optional) message that prompts the user to respond to the items listed. When all responses are made, the user presses **Go** to exit the form. Exit responses must be handled by the programmer using FormEdit. For example, if the user doesn't press the correct key to exit, the program might respond by beeping.

In Figure 3-1, the response field 'User Name' requires that the user enter a string. Note that when the form is first displayed (as shown), the default name 'Margaret' is displayed. The user can overtype this string with a different string response. Instead of displaying the contents of a field, you can elect to display each character the user types with the pound sign character (#) rather than with the characters actually typed.

In Figure 3-2, the user is required to select either either Yes or No by highlighting the appropriate choice. The default value for this field is highlighted in the example. The user can change this value by highlighting the other choice.

For each field highlighted in a form, a message prompting the user is displayed in the area just below the fields. The prompt message describes exactly how the user is to respond to the field highlighted. In Figure 3-1, the field 'User Name' is highlighted, and the following prompt is displayed:

```
Enter the user name you wish to use
```

Using the **Next** key, the user can highlight other fields in a form. For each field, a different prompt message can be displayed.

For each field in a form, the user can optionally be provided online help. Say, for example, a message file is used to display a message for the 'User Name' field shown in Figure 3-1. To obtain more information on how to respond to the user name, the user highlights the field 'User Name' and presses **Help**. By doing so, a help message and an exit message, such as those shown below, are displayed:

```
Type in your first name.

Press Cancel to exit.
```

FormEdit calls the ZoomBox operation to create a box that surrounds these messages. (ZoomBox is called automatically by FormEdit whenever the user presses **Help**. See the description of ZoomBox for details.)

A title appears on the top border of a help box. For each field in a form created by FormEdit, you should provide a unique help box title.

The help message appears in the middle of the box. In the previous example, the message is

Type in your first name.

The help message displayed is followed by a prompt instructing the user on how to exit the help box. In all cases, activating the **Cancel** key terminates the help box (causes FormEdit to call the UnzoomBox operation). In the example, the prompt message is

Press **Cancel** to exit.

To set up each field in a form and its corresponding help file, the caller provides FormEdit with a parameter descriptor. FormEdit refers to the contents of the descriptor to display the form and process user input. There are a total of four different kinds of descriptors. The type of descriptor depends on whether the field requires a user-entered string or a choice response and whether the caller is using a message file or an in-memory buffer for storing strings. Each field descriptor is an element in the array of field descriptors for this operation. (See the description of *prgpFieldDescs* for the field descriptor formats.)

FormEdit calls the NlsClass operation to determine if the characters used in choices the user can select are alphanumeric (that is, are displayable characters). For choice responses that are Yes or No, FormEdit calls NlsGetYesNoStrings. NlsCase is called to convert the first character of each choice response that is not Yes or No to its uppercase value. For this reason, each choice must start with a unique character. Form processing is based on default values or on nationalizable tables in the Native Language Support (NLS) table area. See the description of *pNlsTableArea* (below). (For details on NLS, see "Native Language Support" in the *CTOS Operating System Concepts Manual*.)

The FormEdit operation is typically used by a software installation program to obtain information from the user about how an installation should be customized. (For details on installation, see the *CTOS Programming Guide*.)

Procedural Interface

FormEdit (pNlsTableArea, iFrame, iCol, iLine, nFields, pbPrompt, cbPrompt, prgpFieldDescs, sMaxFields, maxCaptionLen, pChRet): ercType

where

pNlsTableArea

FormEdit returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, FormEdit processes the form based on default values.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, FormEdit processes the form based on tables in Nls.sys.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, FormEdit processes the form based on the alternate set of ImaNLS tables.

iFrame

is a word value that specifies the frame.

iCol

specifies the horizontal coordinate (word) within *iFrame* at which the first character of the form is to be displayed.

iLine

specifies the vertical coordinate (word) within *iFrame* at which the first character of the form is to be displayed.

nFields

specifies the number of fields (word) in the form. For the form shown in Figure 3-2, *nFields* is 5.

pbPrompt
cbPrompt

describe an (optional) prompt displayed within the form. The prompt can be used, for example, to provide the user with general instructions on how to fill out the form and to indicate which key the user must press to exit the form. The example forms shown in Figure 3-1 and 3-2 display the following prompt:

```
Fill in or modify, press GO to accept, CANCEL to dismiss.
```

If *cbPrompt* > 0, the prompt is the first string displayed in the form.

prgpFieldDescs

is the memory address of an array of pointers to field descriptors. Each field descriptor describes an individual field in the form. There are a total of four different field descriptors. Basically, one type of descriptor is used for string responses and the other, for user-selected choices. In addition, for each of these descriptor types, there are two versions: one is used with message files and the other, with message strings stored in memory.

The format of each of these descriptor types is described below:

String Responses Using Strings In Message Files

Field	Size (Bytes)	Description
bFieldType	1	determines how the user response is displayed in the field. 0 means display the characters the user types. 1 means display a pound sign (#) for each character typed.
sFieldDesc	2	is 27 (the size in bytes of this descriptor).
iMsgCaption	2	is the message file number of the field caption string, such as 'User Name' in Figure 3-1.
iMsgDefault	2	is the message file number of the default string shown in the field when the form is first invoked. In the field 'User Name' in Figure 3-1, the name 'Margaret' is displayed as the default string.
iMsgPrompt	2	is the message number of the prompt string displayed at the bottom of the form for this field. For example, when the field 'User Name' is highlighted in Figure 3-1, the prompt 'Enter the user name you wish to use' is displayed
pbRet	4	is the memory address at which the user-entered string is returned.
cbRet	2	is the size in bytes of the returned string.

String Responses Using Strings In Message Files (*continued*)

Field	Size (Bytes)	Description
pHelpProc	4	is the memory address of a user-written help procedure called by FormEdit or is 0. If *pHelpProc* is not 0, *pHelpProc* is the address of a user procedure of the following format: Call Proc(*wHelpToken*, *pErcRet*) *wHelpToken* is described in the next field of this descriptor. (See *wHelpToken* below.) *pErcRet* is the address of a status code returned indirectly to FormEdit. If, however, *pHelpProc* is 0, a help message is displayed to the screen from a help message file. The fields for the help messages are *iMsgHelp*, *iMsgHelpPrompt*, and *iMsgHelpTitle*. (These fields are described below.)
wHelpToken	2	is a number passed to the user-written help procedure for use in displaying a custom help menu.
iMsgHelp	2	is the message number of the help message to be displayed in the middle of the help box.
iMsgHelpPrompt	2	is message number of the prompt message to be displayed below the help message in the help box.

String Responses Using Strings In Message Files (*continued*)

Field	Size (Bytes)	Description
iMsgHelpTitle	2	is message number of a help title to be displayed on the top border of the help box. When the ZoomBox operation creates the box surrounding the help message, the help title appears on the top border. (See ZoomBox for details.)

String Responses Using Strings Stored In Memory

Field	Size (Bytes)	Description
bFieldType	1	determines how the user response is displayed in the field. 200 means display the characters the user types. 201 means display a pound sign (#) for each character typed.
sFieldDesc	2	is 39 (the size in bytes of this descriptor).
pbCaption	4	is the memory address of the field caption string, such as 'User Name' in Figure 3-1. (The string is stored in memory rather in a message file.)
cbCaption	2	is the size of the field caption.
pbDefaultText	4	is the memory address of the default string shown in the field when the form is first invoked. In the field 'User Name' in Figure 3-1, the name 'Margaret' is displayed as the default string.

Choice Responses Using Strings In Message Files

Field	Size (Bytes)	Description
cbDefaultText	2	is the size in bytes of the default string.
pbRet	4	is the memory address at which the user-entered string is returned.
cbRet	2	is the size in bytes of the returned string.
pbPrompt	4	is the memory address of the prompt string displayed at the bottom of the form for this field. For instance, when the field 'User Name' is highlighted in Figure 3-1, the prompt 'Enter the user name you wish to use' is displayed.
cbPrompt	2	is the size in bytes of the prompt string.
pHelpProc	4	This field and the four that follow it are the same help fields as described previously in "String Responses Using Strings In Message Files."
wHelpToken		See *pHelpProc*.
iMsgHelp		See *pHelpProc*.
iMsgHelpPrompt		See *pHelpProc*.
iMsgHelpTitle		See *pHelpProc*.

Choice Responses Using Strings In Message Files (*continued*)

Field	Size (Bytes)	Description
bFieldType	1	determines the type of response prompted by this field. 2 means the user selects a response from choices that are not 'Yes' or 'No'. 3 means the user selects 'Yes' or 'No.'
sFieldDesc	2	is 35 (the size in bytes of this descriptor).
imsgCaption	2	is the message file number of the field caption string, such as 'Save copy of backup' in Figure 3-2.
prgImsgChoices	4	is the memory address of an array of message numbers of possible responses to this field. The user selects a response by highlighting the appropriate response using the **Right Arrow** or **Left Arrow** key.
nChoices	1	is the number of choices for this field, for example, the number of choices for the field 'Save copy of backup' in Figure 3-2' is 2 (Yes or No).

Choice Responses Using Strings In Message Files (*continued*)

Field	Size (Bytes)	Description
iDefault	1	is the number of the response item that is highlighted by default. From left to right, numbering is consecutive starting with 0. In Figure 3-2, 'Yes' is 0 and 'No' is 1. If 'Yes' is highlighted by default, *iDefault* is 0.
iMsgPrompt	2	is the message number of the prompt string displayed at the bottom of the form for this field. For instance, when the field 'Public' is highlighted in Figure 3-2, the message number of the prompt string 'Press Y or N, or use arrow keys' is displayed.
pChoiceRet	4	is the memory address of a byte whose value indicates the message selected by the user. If the user selected 'No' for the field 'Public' in Figure 3-2, the value 1 would be returned. (0 would be returned if 'Yes' were selected.)
pWorkArea	4	is the memory address of an internal work area used by FormEdit for this field.
sWorkArea	2	is the work area size in bytes. The minimum size is 150 bytes plus 5 bytes per choice. The minimum size for the field' Public' in Figure 3-2 is 160 bytes (150 bytes plus 5 bytes for each of the two choices 'Yes' and 'No').

Choice Responses Using Strings In Message Files (*continued*)

Field	Size (Bytes)	Description
pHelpProc	4	This field and the four that follow it are the same help fields as described previously in "String Responses Using Strings In Message Files."
wHelpToken	2	See *pHelpProc*.
iMsgHelp	2	See *pHelpProc*.
iMsgHelpPrompt	2	See *pHelpProc*.
iMsgHelpTitle	2	See *pHelpProc*.

Choice Responses Using Strings Stored In Memory

Field	Size (Bytes)	Description
bFieldType	1	determines the type of response prompted by this field. 202 means the user selects a response from choices that are not 'Yes' or 'No.' 203 means the user selects 'Yes' or 'No.'
sFieldDesc	2	is 43 (the size in bytes of this descriptor).
pbCaption	4	is the memory address of the field caption string, such as 'Public' in Figure 3-2.
cbCaption	2	is the count of bytes in the caption, for example, 6 for the caption 'Public'
prgSdChoices	4	is the memory address of an array of Sdtype (6-byte) records describing the choices that can be selected. For each record, the first 4 bytes contain the memory address of the choice string, and the last 2 contain the string byte count. If, for example, 'Yes' is one of the choices, *prgSdChoices* would contain an Sdtype record of the following form for this choice: a 4-byte pointer to the string 'Yes' a 2-byte word containing the byte count value of 3

Choice Responses Using Strings Stored In Memory (*continued*)

Field	Size (Bytes)	Description
nChoices	1	is the number of choices for this field, for instance, the number of choices for the field 'Public' in Figure 3-2 is 2 ('Yes' or 'No').
iDefault	1	is the number of the response item that is highlighted by default. From left to right, numbering is consecutive starting with 0. In Figure 3-2, 'Yes' is 0 and 'No' is 1. If 'Yes' is highlighted by default, *iDefault* is 0.
pbPrompt	4	is the memory address of the prompt string displayed at the bottom of the form for this field. For instance, if the field 'Public' is highlighted in Figure 3-2, the string 'Press Y or N, or use arrow keys' is displayed.
cbPrompt	2	is the byte count of the prompt string.
pChoiceRet	4	is the memory address of the message selected by the user. If the user selected 'No' for 'Public' in Figure 3-2, the value 1 would be returned. (0 would be returned if 'Yes' were selected.)
pWorkArea	4	is the memory address of an internal work area used by FormEdit for this field.

Choice Responses Using Strings Stored In Memory (*continued*)

Field	Size (Bytes)	Description
sWorkArea	2	is the work area size in bytes.
pHelpProc	4	This field and the four that follow it are the same help fields as described previously in "String Responses Using Strings In Message Files."
wHelpToken	2	See *pHelpProc*.
iMsgHelp	2	See *pHelpProc*.
iMsgHelpPrompt	2	See *pHelpProc*.
iMsgHelpTitle	2	See *pHelpProc*.

sMaxFields

is the size (word) in bytes of the largest string response for this form. If, for example, *sMaxFields* were 10 and the user entered a string of 12 bytes, only the first 10 bytes would be displayed.

maxCaptionLen

is the size (word) in bytes of the largest caption that is part of the form. In Figure 3-2, *maxCaptionLen* would be 26 (the count of bytes in the field 'Save defaults in user file').

pChRet

is the memory address of the byte where the exit character is to be written. The exit character is used to determine if the menu should be exited (such as when the user presses **Go** in the example form) or if some other appropriate action should be taken.

Request Block

FormEdit is an object module procedure.

This page intentionally left blank.

ForwardRequest (exch, prq): ercType

Description

The ForwardRequest primitive is used by filter processes to forward a request block to another system service for further processing. It should be used only by filter processes that intercept a request on its way to the original system service, but not on its return trip to the client.

In real mode, filter processes of this sort can use the send or RequestDirect primitive to forward the request to the original system service.

To make such a program work in protected mode, use ForwardRequest in place of Send or RequestDirect. ForwardRequest is similar to Send in that no response is expected; by issuing it, the filter process discharges its responsibility for the request as if it had done a respond. ForwardRequest is similar to RequestDirect in that its argument must always be the memory address of a request block.

Procedural Interface

ForwardRequest (exch, prq): ercType

where

exch

is the exchange where the request block is to be sent.

prq

is the memory address of the request block.

Request Block

ForwardRequest is a Kernel primitive.

This page intentionally left blank

FProcessorSupportsProtectedMode: flagType

Description

FProcessorSupportsProtectedMode returns TRUE on an 80286 or subsequent microprocessor (a processor capable of protected mode execution).

FProcessorSupportsProtectedMode returns FALSE on an 8086 or 80186 microprocessor.

The actual mode in which the processor is executing (real or protected) has no effect on the result.

Procedural Interface

FProcessorSupportsProtectedMode: flagType

Request Block

FProcessorSupportsProtectedMode is an object module procedure.

This page intentionally left blank

FProtectedMode: flagType

Description

FProtectedMode returns TRUE if the calling program is executing in protected mode. It returns FALSE if the program is executing in real mode.

FProtectedMode may be used on any Intel microprocessor. On microprocessors, such as the 8086 and the 80186, which do not support protected mode execution, FProtectedMode always returns FALSE.

Procedural Interface

FProtectedMode: flagType

Request Block

FProtectedMode is an object module procedure.

This page intentionally left blank

FrameBackSpace (iFrame, pFrame, piCol, piLine): ercType

Description

FrameBackSpace searches backwards from the location specified by the frame coordinates in the specified frame to find the first non-null character. Then, it updates the frame coordinates to reflect the character following the non-null character.

This operation is used by video byte streams, for example, to locate the next position within *iFrame* at which to begin writing to the screen.

Procedural Interface

FrameBackSpace (iFrame, pFrame, piCol, piLine): ercType

where

iFrame

specifies the index of the frame descriptor in the Video Control Block. An index value of 255 is treated as a special case and is reserved for internal use only.

pFrame

is the memory address of the frame descriptor.

piCol

is the memory address of a byte containing the horizontal coordinate.

piLine

is the memory address of a byte containing the vertical coordinate.

Request Block

FrameBackSpace is a system-common procedure.

FRmos: flagType

Description

FRmos returns TRUE if the calling program is executing in real mode and the operating system is executing in protected mode.

Procedural Interface

FRmos: flagType

Request Block

FRmos is an object module procedure.

This page intentionally left blank

FsCanon (pbFileSpec, cbFileSpec)

NOTE: This operation does not return a status code.

Description

FsCanon performs a character-by-character translation of the file specification into the file system canonical form.

Procedural Interface

FsCanon (pbFileSpec, cbFileSpec)

where

pbFileSpec
cbFileSpec

describe the file specification to be translated.

Request Block

FsCanon is a system common procedure.

This page intentionally left blank

FSrpUp: flagType

Description

FSrpUp returns TRUE if the cluster is operating.

Procedural Interface

FSrpUp: flagType

Request Block

FSrpUp is a system common procedure.

This page intentionally left blank

FValidPbCb (pb, cb): ercType

Description

FValidPbCb verifies that the data area specified is accessible by the calling program.

When the calling program is running in protected mode, the access rights and length of the selector are checked. In real mode, the only test made is that the area specified doesn't wrap around (the length isn't greater than the difference between the current offset and the maximum segment length of 65,535 bytes).

Procedural Interface

FValidPbCb (pb, cb): ercType

where

pb
cb

describes the memory to be verified.

Request Block

FValidPbCb is an object module procedure.

This page intentionally left blank

FVSeries: flagType

Description

FVSeries returns TRUE if the calling program is executing in protected mode on a virtual memory operating system. FVSeries can be used on any Intel microprocessor. On microprocessors such as the 8086, 80186, and 80286 (which do not support virtual memory management), FVSeries always returns FALSE.

Procedural Interface

FVSeries: flagType

Request Block

FVSeries is an object module procedure.

This page intentionally left blank.

GenResString (pStringRet, cStringMax, psStringRet): ercType

Description

GenResString generates an ASCII string that describes the resolution of the caller's monitor. This operation is useful for generating names of font files containing data valid to only one monitor type. The string returned is of the form

HorizontalResolutionXVerticalResolution

For example, a string returned by GenResString when run on a system with a VC003/GC004 would appear as "1024X768." When GenResString is called on a machine that does not have a graphics monitor, *sStringRet* is 0.

Procedural Interface

GenResString (pStringRet, cStringMax, psStringRet): ercType

where

pStringRet
cStringMax

describe the area into which GenResString will place the string. *pStringRet* is the memory address at which the string will start. *cStringMax* (word) is the maximum string size.

psStringRet

is the memory address of a word into which GenResString will place the actual length of the string.

Request Block

GenResString is an object module procedure.

GetAltMsg (pMwa, sMwa, iMsg, pRgSd, sRgSd, pBuffer, sBuffer, pcbRet): ercType

Description

GetAltMsg retrieves a message from a message file opened by calling the InitAltMsgFile operation and places the expanded message in memory. If the message contains macros, the macros are expanded with the corresponding string descriptor. (See Appendix F for a description of the message file macros.)

GetAltMsg is similar to GetMsg except that, with alternate message files, any number of message files may be open at any time. Access to each message file is determined by the Message Work Area (MWA).

Procedural Interface

GetAltMsg (pMwa, sMwa, iMsg, pRgSd, sRgSd, pBuffer, sBuffer, pcbRet): ercType

where

pMwa

is the memory address of the same MWA that was supplied to InitAltMsgFile.

sMwa

is the size of the MWA. The size must be at least 60 bytes.

iMsg

is the message number.

pRgSd
sRgSd

is the memory address of an array of string descriptors. Each element of the array describes a string that is inserted into the message in place of a macro. String descriptors are 6 byte blocks of memory. The first 4 bytes contain the memory addressof the string. The last 2 bytes contain the string size. (See Appendix F for details on macros.)

pBuffer

is the memory address of a buffer to which the expanded message is written.

sBuffer

is the size of the buffer.

pcbRet

is the memory address of a word to which the actual length of the expanded message is returned.

Request Block

GetAltMsg is an object module procedure.

GetAltMsgUnexpanded (pMwa, sMwa, iMsg, pBuffer, sBuffer, pcbRet):
ercType

Description

GetAltMsgUnexpanded retrieves a message from a message file opened by calling the InitAltMsgFile operation and places the unexpanded message in memory, that is, it brings in an exact copy of the message text without expanding any macros which may be present. This operation generally is used in conjunction with the related operation, ExpandLocalMsg.

GetAltMsgUnexpanded is similar to GetMsgUnexpanded except that, alternate message files, any number of message files may be open at any time. Access to each message file is determined by the Message Work Area (MWA).

Procedural Interface

GetAltMsgUnexpanded (pMwa, sMwa, iMsg, pBuffer, sBuffer, pcbRet):
ercType

where

pMwa

is the memory address of the same MWA that was supplied to InitAltMsgFile.

sMwa

is the size (word) of the MWA. The size must be at least 60 bytes.

iMsg

is the message number (word).

pBuffer
sBuffer

describe the buffer to which the expanded message is written.

pcbRet

is the memory address of a word to which the actual length of the expanded message is returned.

Request Block

GetAltMsgUnexpanded is an object module procedure.

GetAltMsgUnexpandedLength (pMwa, sMwa, iMsg, pcbRet): ercType

Description

GetAltMsgUnexpandedLength returns the length of a message in its unexpanded form. This operation generally is used to determine the memory needed to allocate for a message.

GetAltMsgUnexpandedLength is similar to GetMsgUnexpandedLength, except that, with alternate message files, any number of message files may be open at any time. Access to each message file is determined by the Message Work Area (MWA).

Procedural Interface

GetAltMsgUnexpandedLength (pMwa, sMwa, iMsg, pcbRet): ercType

where

pMwa

is the memory address of the same MWA that was supplied to InitAltMsgFile.

sMwa

is the size of the MWA (word). The size must be at least 60 bytes.

iMsg

is the message number (word).

pcbRet

is the memory address of a word to which the length of the unexpanded message is returned.

Request Block

GetAltMsgUnexpandedLength is an object module procedure.

GetBoardInfo (wSlot, wCase, pbInfoRet, cbInfoMax): ercType

Caution: *This operation works on shared resource processor hardware only. Do not use it on workstation hardware.*

NOTE: Programs that may run on earlier versions of the operating system should use this object module procedure rather than the system-common procedure QueryBoardInfo, which returns the same information.

Description

GetBoardInfo accepts the slot number of an SRP processor board and a case value. Based on the case value specified, information is returned about the specified processor in the memory area provided. If the memory area is too small to contain all the information returned by this operation, the information is truncated.

Procedural Interface

GetBoardInfo (wSlot, wCase, pbInfoRet, cbInfoMax): ercType

where

wSlot

is the slot number (word) of the processor. (For details on shared resource processor slot number naming conventions, see the *CTOS System Administration Guide*.)

wCase

is a word value that is currently 0. The structure returned by case 0 contains the following information for the processor specified by *wSlot*:

Offset	Field	Size (bytes)	Description
0	localUserNumMax	2	Is the maximum number of local user numbers.
2	iCpu	2	Is the CPU number. (See the description of *iCpu* in the structure returned by the GetUserLocation operation.)
4	fMfProcessor	1	Is a flag that is TRUE if the processor is the master processor.

pbInfoRet
cbInfoMax

describe the memory area to which the structure specified by *wCase* is returned.

Request Block

GetBoardInfo is an object module procedure.

GetBsLfa (pBswa, pLfaRet): ercType

Description

GetBsLfa returns the logical file address at which the next input/output operation will occur for the open byte stream identified by the memory address of the Byte Stream Work Area.

Procedural Interface

GetBsLfa (pBswa, pLfaRet): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pLfaRet

is the memory address to which the current logical file address is to be returned.

Request Block

GetBsLfa is an object module procedure.

This page intentionally left blank

GetBsMember (pBSWA, memberCode, pStructRet, sStructMax, pcbStructRet): ercType

Description

GetBsMember copies the contents of the specified member (field) in the indicated Byte Stream Work Area (BSWA) to the memory area provided. If the BSWA or the member code specified is invalid, status code 7 ("Not implemented") is returned. It is strongly recommended that applications use this operation rather than directly accessing the contents of a BSWA.

Procedural Interface

GetBsMember (pBSWA, memberCode, pStructRet, sStructMax, pcbStructRet): ercType

where

pBSWA

is the memory address of the same Byte Stream Work Area that was supplied to the OpenByteStream operation.

memberCode

is a code identifying the Byte Stream Work Area member. The valid member codes and their descriptions are

Code	Member	Size (bytes)	Description
0	fh	2	File handle of the file opened by the call to OpenByteStream. The fh is only valid for a Disk, GPAM, or Spooler byte stream.

pStructRet
sStructMax

describe the memory area to which the specified member information is copied. This memory area must be large enough to accommodate the contents of the BSWA member, or the data is truncated.

pcbStructRet

is the memory address of a word to which the number of bytes of BSWA member information successfully copied is returned.

Request Block

GetBsMember is an object module procedure.

GetCanonicalNodeAndVol (pbNode, cbNode, pbCanonNode, pcbCanonNode, pbVol, cbVol, pbCanonVol, pcbCanonVol): word

Description

GetCanonicalNodeAndVol converts a node and/or volume *token* (that is, node and/or volume name string) into its canonical form. For example, the node token Local might be converted to the node name Pubs, or the device specification d1 might be converted to the volume name Win.

The brackets surrounding the token must be removed before this operation is called. To remove the brackets, your application must call the RemoveFfsBrackets operation. (See the description of RemoveFfsBrackets in this chapter.)

GetCanonicalNodeAndVol performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a 0 value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*. The meanings of the nonzero values returned are listed in Table 11-3.)

Procedural Interface

GetCanonicalNodeAndVol (pbNode, cbNode, pbCanonNode, pcbCanonNode, pbVol, cbVol, pbCanonVol, pcbCanonVol): word

where

pbNode
cbNode

describe the node token.

pbCanonNode

is the memory address at which the node token canonical form is returned.

pcbCanonNode

is the memory address at which the size of the node token canonical form is returned.

pbVol
cbVol

describe the volume token.

pbCanonVol

is the memory address at which the volume token canonical form is returned.

pcbCanonVol

is the memory address at which the size of the volume token canonical form is returned.

Request Block

GetCanonicalNodeAndVol is an object module procedure.

GetClstrGenerationNumber (pbInfoRet): ErcType

Description

GetClstrGenerationNumber returns a generation number that can be used to determine if the cluster has been disconnected. The generation number is incremented each time the cluster workstation is disconnected from the server. The generation number can be used to determine if remote handles are still valid.

Procedural Interface

GetClstrGenerationNumber (pbInfoRet): ErcType

where

pbInfoRet

is the address of the following structure supplied to the user.

Offset	Field	Size (Bytes)	Description
0	wGen	2	The generation number of the current cluster session.
2	fServerUp	1	Flag indicating if the server is presently communicating with the cluster.
3	Reserved	1	Reserved for future use.

Request Block

GetClstrGenerationNumber is a system common procedure.

This page intentionally left blank.

GetClusterId (pRq, pbRet): ercType

Description

GetClusterId returns the identification number of the workstation upon which the client whose request block is at *pRq* is executing. In addition, the following hardware configuration information for the workstation is returned:

- memory size in 32K byte blocks
- X–bus module IDs for bus positions 1 to 8
- I-bus module IDs for I-bus postions 1 to 5

If no module exists for a given bus position, 0 is returned.

This operation is similar to the GetDaiNumber operation, which is only supported on CTOS systems.

Procedural Interface

GetClusterId (pRq, pbRet): ercType

where

pRq

is the memory address of the request block of the client whose workstation hardware is being queried.

pbRet

is the memory address of a 32-byte structure into which the ID and hardware configuration information is copied. The format of this structure is shown below:

Offset	Field	Size (bytes)	Description
0	hwId	2	Hardware ID of the cluster workstation
2	clMemSize	2	Memory of the cluster workstation in 32K byte blocks
6	xBusModIds	16	Module Ids of the eight X-Bus module positions. The Ids are 2 byte quantities and start with the first module attached to the cluster workstation, as described below:

Offset	Size	Description
0	2	First module Id
2	2	Second module Id
4	2	Third module Id
6	2	Fourth module Id
8	2	Fifth module Id
10	2	Sixth module Id
12	2	Seventh module Id
14	2	Eighth module Id

Offset	Field	Size (bytes)	Description
22	iBusModIds	10	Module Ids of the five I-Bus modules connected to the cluster workstation. The Ids are 2 byte quantities and start with the first module attached to the cluster workstation, as described below:

Offset	Size	Description
0	2	First module Id
2	2	Second module Id
4	2	Third module Id
6	2	Fourth module Id
8	2	Fifth module Id

Request Block

GetClusterId is a system-common procedure.

This page intentionally left blank

GetClusterStatus (statParam, pBufferRet, sBufferMax): ercType

Description

GetClusterStatus returns usage statistics for each communications line and for the workstations attached to it. Statistics are returned in the Communications Status Buffer structure. For the format of this structure, see "Communications Status Buffer" in Chapter 4, "System Structures."

Procedural Interface

GetClusterStatus (statParam, pBufferRet, sBufferMax): ercType

where

statParam

is a word consisting of 2 byte fields: the low-order byte field specifies the communications line (*iLine*) and the high-order field, the workstation identification number (*wsId*).

Master usage statistics for the specified communications line are returned if all the bits in the *wsId* byte are 0. Otherwise, usage statistics are returned for the workstation identified by *wsId*.

The *iLine* codes and their descriptions are as follows:

Code	Description
128	The single RS-422 line on a server workstation operating systems prior to CTOS/VM 2.2. (See OsVersion or CurrentOsVersion.)
129	The first line on a server workstation (CTOS/VM 2.2 and later operating systems) or on the first Cluster Processor (CP) on a shared resource processor server.
130	The second line on the first CP on a shared resource processor server.
131	The first line on the second CP on a shared resource processor server.
132	The second line on the second CP on a shared resource processor server.
.	
.	
.	
	and so on for each additional CP.

pBufferRet
sBufferMax

describe the buffer to which the Communications Status Buffer is placed. If *sBufferMax* is too small, the statistics are truncated.

If you are specifying the size of cluster workstation statistics, the minimum size of *sBufferMax* is 176 bytes (that is, 32 bytes plus 144 bytes for the Drop Control Table). (See "Communications Status Buffer" in Chapter 4, "System Structures.")

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	100
12	statparam	2	
14	reserved	4	
18	pBufferRet	4	
22	sBufferMax	2	

This page intentionally left blank

GetCommLineDmaStatus (commlinehandle, pbStatus, cbStatus): ercType

Description

GetCommLineDmaStatus returns the number of bytes transferred during the current CommLineDMA operations (TransmitCommLineDma or ReceiveCommLineDma) for both the transmit and receive DMA channels.

Because GetCommLineDmaStatus is a system common procedure, it may be called from a communications interrupt handler.

Procedural Interface

GetCommLineDmaStatus (commlinehandle, pbStatus, cbStatus): ercType

where

commLineHandle

is a handle returned by InitCommLine.

pbStatus
cbStatus

describe the status information to be returned. The format of the status area is as follows:

Offset	Size	Description
0	2	count of bytes already received.
2	2	count of bytes already transmitted.

Request Block

GetCommLineDmaStatus is a system common procedure.

This page intentionally left blank

GetCoprocessorStatus (pStatusRet): ercType

Description

GetCoprocessorStatus reports if either a numeric coprocessor or a numeric coprocessor emulator (such as the Math Service) is present. A coprocessor or emulator is required to execute floating point instructions.

To query the presence of a numeric coprocessor without regard to an emulator, use the QueryCoprocessor operation.

Procedural Interface

GetCoprocessorStatus (pStatusRet): ercType

where

pStatusRet

is the memory address of a structure into which information concerning the coprocessor is written. The structure and values are as follows:

Offset	Field	Size (bytes)	Description
0	processorType	2	0 = unknown 1 = 8086 2 = 80186 3 = 80286 4 = 80386
2	mathServiceInfo	2	Machine status word on an 80286- or 80386-based microprocessor used by the Math Service during installation.
4	fCoprocessorPresent	1	If TRUE, either a numeric coprocessor or an emulator is present. Otherwise, neither is present.

Request Block

GetCoprocessorStatus is an object module procedure.

GetCParasOvlyZone: WORD

Description

GetCParasOvlyZone returns the size of the overlay buffer measured in paragraphs. A paragraph is 16 bytes.

Procedural Interface

GetCParasOvlyZone: WORD

where the returned word contains the size of the overlay buffer measured in paragraphs.

Request Block

GetCParasOvlyZone is an object module procedure.

This page intentionally left blank

GetCtosDiskPartition (fh, pOffsetRet, pPartitionSize): ercType

Description

GetCtosDiskPartition returns the disk offset and size of the CTOS partition on a partitioned hard disk. Using GetCtosDiskPartition, the calling program can determine the location of the starting Volume Home Block on the partitioned disk. Note that on a partitioned hard disk, the starting Volume Home Block does not always reside in the first 14 sectors.

Before an application calls GetCtosDiskPartition, it must first access the disk device by using the OpenFile operation. In the call to OpenFile, the application must provide a device or volume specification (for example, [d0]). OpenFile then returns a handle that can be used in a subsequent call to GetCtosDiskPartition.

On a partitioned hard disk, CTOS files and DOS files can each reside in a separate disk partition. Note that partitioned hard disks are only supported on SuperGen Series 5000 workstations. If the application uses GetCtosDiskPartition to access a nonpartitioned disk, GetCtosDiskPartition returns a value of 0 at *pPartitionSize*.

For details on disk partitions, see "Disk Management" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

GetCtosDiskPartition (fh, pOffsetRet, pPartitionSize): ercType

where

fh

is the handle returned by the call to OpenFile.

pOffsetRet

is the memory address of a double word to which the disk offset of the CTOS disk partition is returned.

pPartitionSize

is the memory address of a double word to which the size of the CTOS disk partition is returned.

Request Block

GetCtosDiskPartition is an object module procedure.

GetDateTime (pDateTimeRet): ercType

Description

GetDateTime returns a pointer to a 4-byte structure that contains the current date and time in the operating system date/time format. (For a description of this format, see "System Date/Time Structure" in Chapter 4, "System Structures.") The structure returned by this call consists of the *seconds* field and the *dayTimes2* field of the system date/time structure.

The system date/time format is represented in 32 bits to an accuracy of 1 second. The high-order 15 bits of the high-order word contain the count of days since March 1, 1952. The use of a 15-bit field allows dates up to the year 2042 to be represented. The low-order bit of the high-order word is 0 to represent AM and 1 to represent PM. The low-order word contains the count of seconds since mignight/noon. Valid values are 0 to 43199.

Status code 46 ("Master workstation going down") is returned if a DisableCluster operation was performed. If status code 46 ("Master workstation going down") is returned, the *seconds* field of the structure pointed to by *pDateTimeRet* contains the time (in seconds) remaining before the server workstation goes down.

NOTE: Status code 46 is only returned if the application sets the fReturnErc *flag in the request block. Since* fReturnErc *is not a parameter in the procedural interface, the application must build a request block to set this flag. The application can then issue the GetDateTime request by calling Request, RequestDirect, or RequestRemote.*

Procedural Interface

GetDateTime (pDateTimeRet): ercType

where

pDateTimeRet

is the memory address of a 4-byte block to which the date and time in the System Date/Time format is returned.

Request Block

sDateTimeRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	14
12	fReturnErc	1	
13	reserved	5	
18	pDateTimeRet	4	
22	sDateTimeRet	2	4

GetDirInfo (pbDirectorySpec, cbDirectorySpec, pbDirectoryPW, cbDirectoryPW, pDirectoryEntryRet, sDirectoryEntryMax): ercType

Description

GetDirInfo copies the requested directory entry from the Master File Directory to the specified memory area. If the specified area is not large enough to hold all of the requested information, the information is truncated. (For the structure of a directory entry, see "Entry for a Directory in a Master File Directory Sector" in Chapter 4, "System Structures.")

Procedural Interface

GetDirInfo (pbDirectorySpec, cbDirectorySpec, pbDirectoryPW, cbDirectoryPW, pDirectoryEntryRet, sDirectoryEntryMax): ercType

where

pbDirectorySpec
cbDirectorySpec

describe the directory specification to search for in the Master File Directory. Examples are

[Sys]<Tmp>
{Os}[d1]<Tmp>
Tmp

pbDirectoryPW
cbDirectoryPW

describe the password of the specified directory.

pDirectoryEntryRet
sDirectoryEntryMax

describe the area where the directory entry is returned.

Request Block

GetDirInfo is an object module procedure.

GetDirStatus (pbDirName, cbDirName, pbPassword, cbPassword, statusCode, pStatusRet, sStatusMax) : ercType

Description

GetDirStatus allows you to determine information about a directory. Either a volume password or the directory password is required.

Status code 218 ("Bad Mode") is returned if the statusCode is invalid.

Procedural Interface

GetDirStatus (pbDirName, cbDirName, pbPassword, cbPassword, statusCode, pStatusRet, sStatusMax) : ercType

where

pbDirName
cbDirName

describe the directory name.

pbPassword
cbPassword

describe a password that gives access to the directory.

statusCode

specifies the status item. Status items and their codes are as follows:

Code	Item	Size (Bytes)
0	Directory Size	4
1	invalid	
2	File Protection Level	2
3	Password*	13

*The first byte of a password item is the number (0-12) of characters in the password.

pStatusRet
sStatusMax

describe the memory area to which the status information is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	2	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	216
12	statusCode	2	
14	reserved	4	
18	pbDirName	4	
22	cbDirName	2	
24	pbPassword	4	
28	cbPassword	2	
30	pStatusRet	4	
32	sStatusMax	2	

GetErc (erc, pBuffer, sBuffer, pcbRet): ercType

Description

GetErc retrieves an error message from an error message file and places the message in memory.

GetErc is similar to GetMsgUnexpanded except that, with GetErc, the messages are retrieved from the error message file [Sys]<Sys>ErcMsg.bin (previously opened by calling InitErcFile) rather than from the application's normal message file. This allows access to an error file without disturbing the application's message file.

If the error message file has not been previously opened, GetErc will attempt to open the file and allocate the minimum memory needed to access it. If the application has previously allocated all available partition memory, the application must call InitErcMsg before calling GetErc.

Procedural Interface

GetErc (erc, pBuffer, sBuffer, pcbRet): ercType

where

erc

is the status code (word).

pBuffer
sBuffer

describe the buffer to which the expanded message is written.

pcbRet

is the memory address of a word to which the actual length of the message is returned.

Request Block

GetErc is an object module procedure.

GetErcLength (erc, pcbRet): ercType

Description

GetErcLength returns the length of an error message. This operation generally is used to determine the memory needed to allocate for the error text.

Procedural Interface

GetErcLength (erc, pcbRet): ercType

where

erc

is the status code (word).

pcbRet

is the memory address of a word to which the length of the unexpanded message is returned.

Request Block

GetErcLength is an object module procedure.

This page intentionally left blank

GetFhLongevity (fh, pCodeRet): ercType

Description

GetFhLongevity copies the requested information on whether the file handle is long-lived or short-lived to the specified area.

Procedural Interface

GetFhLongevity (fh, pCodeRet): ercType

where

fh

is the file handle returned from an OpenFile operation. The file can be open in either read or modify mode.

pCodeRet

is the memory address of the byte where the longevity code is returned. If the code is 0, the file handle is short-lived; if it is 1, the file handle is long-lived.

Request Block

sCodeRet is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	31
12	fh	2	
14	reserved	4	
18	pCodeRet	4	
22	sCodeRet	2	1

GetFileErc (erc, pbFileName, cbFileName, pBuffer, sBuffer, pcbRet):
ercType

Description

GetFileErc retrieves an error message from a message file and places the error message in memory. GetFileErc is similar to GetErc except that, with GetFileErc, the error text is formatted as

FileName: *Text*

where *FileName* is a text string to be inserted before the error text, and *Text* is the error text. Note that *pbFileName* is not limited to file specifications but may point to any text to be placed before the error message.

Procedural Interface

GetFileErc (erc, pbFileName, cbFileName, pBuffer, sBuffer, pcbRet):
ercType

where

erc

is the status code (word).

pbFileName
cbFileName

describe the string (normally a file name) to be inserted before the error text.

pBuffer
sBuffer

describe the buffer to which the expanded message is written.

pcbRet

is the memory address of a word to which the actual length of the message is returned.

Request Block

GetFileErc is an object module procedure.

GetFileInfoByName (pbSpec, cbSpec, pbPassword, cbPassword, fLocalOnly, specType, pbInfoRet, cbInfoRet): ercType

Description

GetFileInfoByName provides the capability of determining the canonical file specification and device name given a user file specification without opening a file. This request is useful for programs that need to create fully qualified file specifications. Creating canonical file specifications allows programs to pass context-insensitive file specifications to other programs.

For example, Program A has a default path of [sys]<sys> and Program B has a default path of [scr]<$>. If program A passes the file specification 'Memo' to Program B, the two programs will be speaking of two different files. If Program A first builds a canonical specification of [sys]<sys>Memo, the contextual part of the file specification is removed. This allows Programs A and B to talk about the same file.

Procedural Interface

GetFileInfoByName (pbSpec, cbSpec, pbPassword, cbPassword, fLocalOnly, specType, pbInfoRet, cbInfoRet): ercType

where

pbSpec
cbSpec

describe a character string containing a file specification.

pbPassword
cbPassword

describe the password if one is required.

fLocalOnly

can have the following values:

Value	Description
TRUE	The request is not forwarded to the server if the corresponding volume is not local.
FALSE	The request is forwarded to the server if the corresponding volume is not local.

specType

can have the following values:

Value	Description
1	volume
2	directory
3	file

pbInfoRet
cbInfoRet

describe the client buffer in which the following information is returned:

```
fileHeaderNum                    WORD
expanded Canonical Spec
{
   sbVolumeName        [13]      BYTE
   sbDirName           [13]      BYTE
   sbFileName          [51]      BYTE
   sbDeviceName        [13]      BYTE
}
```

maximum size of buffer required = 2+90 = 92 bytes.

For a volume and directory, *fileHeaderNum* returned = 0xFFFF. For a file, the actual *fileHeaderNum* (defined in fileheader.sys) is returned.

The information to be returned is based on the value of *cbInfoRet*. If *cbInfoRet* is less than 92, the information is truncated.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fLocalOnly	2	479
14	specType	2	
16	reserved	2	
18	pbSpec	4	
22	cbSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pbInfoRet	4	
34	cbInfoRet	2	

This page intentionally left blank.

GetFileStatus (fh, statusCode, pStatusRet, sStatusMax): ercType

Description

GetFileStatus copies the requested status information to the specified memory area. If the specified area is not large enough to hold the requested information, the information is truncated.

Procedural Interface

GetFileStatus (fh, statusCode, pStatusRet, sStatusMax): ercType

where

fh

the file handle returned from an OpenFile operation. To get the password, the file must be open in modify mode. If the file is open in read mode, the file password field in the File Header Block is erased. The password fields in the Volume Home Block and Device Control Block are always erased.

statusCode

specifies the status code. The status items and values are as follows:

Code	Item (Bytes)	Size
0	File length	4
1	File class	1
2	File protection level	1
3	Password*	13
4	Date/time of creation	4
5	Date/time last modified	4
6	End-of-file pointer	4
7	Reserved	
8	Volume Home Block	256
9	Device Control Block	100
10	Application-specific field in the File Header Block**	64
11	Node name	13
12	File Header Block+	512

* The first byte of a password item is the number (0-12) of characters in the password.

**For example, this field is used by Word Processor files.

+ This field is valid only on protected mode operating systems.

Code	Item	Size (Bytes)
13	Cache control status	2
14	File removal status	1
15	Internationalization	2

Code 11, Node Name, returns "Local" if the file was opened on the workstation where GetFileStatus was issued. It returns "Master" if the file was opened at a server workstation and GetFileStatus was issued at a cluster workstation connected to that server. It returns the remote node name if the file was opened at a remote node and GetFileStatus was issued at a node other than the one where the file was opened.

Code 13, Cache control status, indicates whether selected files or devices will use the cache. This field is valid only on systems that support the file cacher. (See Appendix A, "Operating System Features," in the *CTOS Operating System Concepts Manual*.) The bits in the word of control information returned have the following meanings:

Bit	Field	Description
0	mCacheDisable	The file has been disabled from caching using the SetFileStatus operation.
1	mCacheEnable	The file has been enabled for caching using SetFileStatus. Asserting *mCacheDisable* and *mCacheEnable* is mutually exclusive. Only one of the bits may be set at a time. If neither is set, cacheability defaults to the state defined by the bits below.
2	mCacheDevDisable	The device upon which the file resides has been disabled from caching using SetFileStatus. Enabling a file for caching (using *mCacheEnable*) overrides disabling the device.

Bit	Field	Description
3	mCacheDevEnable	The device upon which the file resides has been enabled for caching using SetFileStatus. Disabling a file from caching (using *mCacheDisable*) overrides enabling a device.
4	mCacheDefaultEnable	If set, the system default (where the file resides) is to use the cache unless a file or device has been disabled by SetFileStatus. If not set, the system default is not to use the cache unless a file or device has been explicitly enabled for caching.

Code 14, file removal status, indicates whether files should be deleted when file handles are closed. A value of 1 at *pStatusRet* indicates that a file is removed from the directory when a single file handle is closed. It also indicates that the file is removed from the disk when all the file handles are closed. However, the file is only affected if the file handles are closed by CloseFile.

Code 15, internationalization, returns the *wInternational* field of the File Header Block.

pStatusRet
sStatusMax

describe the memory area where the status information is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8
12	fh	2	
14	statusCode	2	
16	reserved	2	
18	pStatusRet	4	
22	sStatusMax	2	

This page intentionally left blank

GetFRmosUser (userNum, pfRmosRet): ercType

Description

GetFRmosUser returns the *fRmosRet* flag. The flag is TRUE if the user number is executing a real mode program.

This operation is used by system services to determine a client's execution mode.

Procedural Interface

GetFRmosUser (userNum, pfRmosRet): ercType

where

userNum

is the user number returned from a CreatePartition or GetPartitionHandle operation. If *userNum* is 0, the *userNum* of the calling program is used.

pfRmosRet

is the memory address of a 1-byte memory area where the *fRmosRet* flag is returned.

Request Block

GetFRmosUser is a system-common procedure.

This page intentionally left blank

GetHandleStatus (pRoute, sRoute, fh, pStatusRet, sStatusRet): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

Description

GetHandleStatus returns statistics on the file represented by the specified file handle.

Procedural Interface

GetHandleStatus (pRoute, sRoute, fh, pStatusRet, sStatusRet): ercType

where

pRoute

is the memory address of the file routing string. The routing string indicates where the file system that originally issued the file handle is installed. The file system could be installed locally or at a remote node, for example.

sRoute

is the size of the routing string.

fh

is the file handle issued by an OpenFile operation.

pStatusRet
sStatusRet

describe the buffer where the routing statistics are placed. If *sStatusRet* is too small, the statistics are truncated.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	297
12	fh	2	
14	reserved	4	
24	pRoute	4	
28	sRoute	2	
18	pStatusRet	4	
22	sStatusRet	2	

GetIBusDevInfo (pInfo, sInfo): ercType

Description

GetIBusDevInfo returns information about the nonkeyboard I-Bus device. The first word of the returned information identifies the device. The remaining information depends on the type of device. If no device is present, wID is set to 0. If a device is present but has not been identified, wID is set to 0FFFFh.

See also PDSetControls, which allows an application to set the speed with which the cursor tracks the mouse, and PSSetSystemcontrols, which allows an application to set the system default speed.

Procedural Interface

GetIBusDevInfo (pInfo, sInfo): ercType

where

pInfo
sInfo

describe the memory area to which the I-Bus device information is copied. If *sInfo* is too small, the data is truncated. The format of the I-Bus information is shown below:

Offset	Field	Size (bytes)	Description
0	wVersion	2	Version of the Mouse Service.
2	wID	2	Identifies the pointing device on the I-Bus. wID = 8 for the 200 dots/inch resolution mouse.
4	wNormMinX	2	Minimum normalized screen coordinates.
6	wNormMinY	2	
8	wNormMaxX	2	Maximum normalized screen coordinates.
10	wNormMaxY	2	
12	fTracking	1	Specifies tracking for the application.
13	bXGear	1	Current x and y gear settings of the application.
14	bYGear	1	
15	bSystemXGear	1	System wide x and y gear settings.
16	bSystemYGear	1	

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32787
12	reserved	6	
18	pInfo	4	
22	sInfo	2	

This page intentionally left blank

GetKbdId (pIdRet): ercType

NOTE: This operation is for backward compatibility only. New applications should use GetKeyboardId.

Description

GetKbdId returns a word value in which the low-order byte is the keyboard identification number (ID) of the attached keyboard.

Procedural Interface

GetKbdId (pIdRet): ercType

where

pIdRet

is the memory address to which the keyboard ID is returned.

Request Block

sIdRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	65524 or -12
12	pIdRet	4	
14	sIdRet	2	2

GetKeyboardId (pIdRet): ercType

Description

GetKeyboardId returns a word value in which the high-order byte is the keyboard identification number (ID) of the attached keyboard.

Procedural Interface

GetKeyboardId (pIdRet): ercType

where

pIdRet

is the memory address to which the keyboard ID is returned.

Request Block

sIdRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	449
12	reserved	6	
18	pIdRet	4	
22	sIdRet	2	2

GetLocalDaiNumber (pDAIRet): ercType

Description

GetLocalDaiNumber returns the Device Address Identification (DAI) number for the workstation on which the caller is running. A DAI number is a physical ID that allows a program to identify a particular workstation among a workstation group, even if all the workstations are running the same operating system.

A specific workstation can be assigned a DAI number either by installing user-designed hardware or by creating the file <Sys>DAI.sys on one of the workstation's hard disks and entering the DAI number in this file. For example, to assign a DAI number of 34 to a workstation, <Sys>DAI.sys should contain the following entry:

```
:DAINumber:34
```

Procedural Interface

GetLocalDaiNumber (pDAIRet): ercType

where

pDAIRet

is the memory address of a 4-byte area in the caller's memory into which the DAI number is returned.

Request Block

GetLocalDaiNumber is a system-common procedure.

This page intentionally left blank

GetMemoryInfo (pbBuffer, cbBuffer, pcbRet): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

Description

GetMemoryInfo returns to the caller information about the state of memory. Such information can include, for example, the upper and lower boundaries of short-lived memory for a particular user number, or the identification of all regions of memory currently allocated.

Procedural Interface

GetMemoryInfo (pbBuffer, cbBuffer, pcbRet): ercType

where

pbBuffer
cbBuffer

describe the buffer to which the information describing the state of memory is placed. (See the Software Release Announcement for the format of this buffer.)

pcbRet

is the memory address of a word to which the count of bytes successfully placed in the buffer is returned.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	306
12	reserved	6	
18	pbBuffer	4	
22	cbBuffer	2	
24	pcbRet	4	
28	scbRet	2	2

GetModuleAddress (iBus, nPos, pIoAddressRet): ercType

NOTE: To ensure portability, applications should use GetModuleAddress instead of calculating base I/O addresses. (See "X-BUS Management" in the CTOS Operating System Concepts Manual.*)*

Description

GetModuleAddress returns the base I/O address of the specified module on the input device bus (I-Bus), system expansion bus (X-Bus), or SuperGen system bus (X-Bus+).

GetModuleAddress is used in conjunction with the GetModuleId operation. Initially, the application calls GetModuleId to determine the position of a specific type of module. The application can then obtain the base address of that module by calling GetModuleAddress with the proper position.

GetModuleAddress returns the same status codes as the GetModuleId operation. (See GetModuleId.)

Procedural Interface

GetModuleAddress (iBus, nPos, pIoAddressRet): ercType

where

iBus

specifies the type of bus and is one of the following values:

Value	Description
1	I-Bus
2	X-Bus
3	X-Bus+

nPos

is a value that specifies the position, on the specified bus, of the module for which to return the base I/O address. The first position is 1, the second position is 2, and so forth.

pModuleIdRet

is the memory address of a word to which the I/O base address of the module is returned.

Request Block

GetModuleAddress is a system-common procedure.

GetModuleId (iBus, nPos, pModuleIdRet): ercType

Description

GetModuleId returns the identification number of the specified module/cartridge on the input device bus (I-Bus), system expansion bus (X-Bus), or SuperGen system bus (X-Bus+).

For I-Bus and X-Bus modules, the high byte of the module ID returned (a word) indicates the module type. The low byte, in turn, indicates module attributes, such as the version number. (See Appendix G for a list of all the I-Bus and X-Bus modules and their IDs.)

For X-Bus+ cartridges, the low word of the ID (a double word) is the manufacturer's code, and the high word is the product ID. For a X-Bus module that has been redesigned as a X-Bus+ cartridge (and is 100% compatible with its X-Bus counterpart), the manufacturer's code is zero, and the high word is of the same format as the X-Bus module ID number.

GetModuleId can be used to list all the modules on a specified bus. When GetModuleId returns status code 35 ("No module in position: *nPos*"), the last value (*nPos*-1) is the number of modules on the bus.

If the current operating system version does not recognize the specified bus, GetModuleId returns status code 36 ("No such bus"). If the specified bus type does not exist on the present workstation, or if no modules/cartridges are connected to the specified bus, GetModuleId returns status code 35 ("No bus or module present").

To determine the position of a particular module, an application can use the QueryModulePosition operation. QueryModulePosition, in turn, repeatedly calls GetModuleId to find the specified X-Bus or I-Bus ID. (See QueryModulePosition.)

The base I/O address of a X-Bus module or a X-Bus+ cartridge can be obtained by calling GetModuleAddress (an object module procedure).

Procedural Interface

GetModuleId (iBus, nPos, pModuleIdRet): ercType

where

iBus

specifies the type of bus and is one of the following values:

Value	Description
1	I-Bus
2	X-Bus
3	X-Bus+

nPos

specifies the module/cartridge position on the bus. This index starts at 1.

pModuleIdRet

is the location of a 2-byte or 4-byte block of memory to which the specified module ID is returned. (See Appendix G.)

Request Block

GetModuleId is a system-common procedure.

GetModuleResourceInfo (wModuleHandle, typeCode, resourceId, pbInfoRet, sInfoRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

GetModuleResourceInfo accesses the specified run file resource at runtime and returns information about it in the caller-provided buffer. The information returned includes the selector addressing the resource and the resource size.

For details on accessing resources in memory, see "Accessing Resources in Memory" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

GetModuleResourceInfo (wModuleHandle, wResourceType, wResourceId, pbInfoRet, sInfoRet): ercType

where

wModuleHandle

is 0 (meaning the current application) or is the DLL handle returned by the LibGetHandle operation.

wResourceType

is the resource type (word).

wResourceId

is the resource ID (word).

pInfoRet

is the memory address of the information returned.

Offset	Field	Size (Bytes)	Description
0	cbInfoRet	2	Is the count of bytes returned in this structure.
2	snResouce	2	Is the selector of the resource in the run file. If *cbResource* is greater than 64K bytes, *snResource* is a 32-bit selector.
4	cbResource	4	Is the count of bytes in the resource.

sInfoRet

is the size of the information returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	443
12	wModuleHandle	2	
14	wResourceType	2	
16	wResourceId	2	
18	pInfoRet	4	
22	sInfoRet	2	

This page intentionally left blank.

GetMsg (iMsg, pRgSd, sRgSd, pbBuffer, sBuffer, pCbRet): ercType

Description

GetMsg retrieves a message from the message file and places the expanded message in memory. If the message contains macros, that is number(s) and/or letter(s) preceded by a percent sign (%), the macros are expanded with the corresponding string descriptor. (See Appendix F for a description of the message file macros.)

GetMsg is similar to PrintMsg, except that PrintMsg places the expanded message in a user-supplied bytestream. (See PrintMsg.)

Procedural Interface

GetMsg (iMsg, pRgSd, sRgSd, pbBuffer, sBuffer, pCbRet): ercType

where

iMsg

is the message number.

pRgSd
sRgSd

is the memory address of an array of string descriptors. Each element of the array describes a string that is inserted in the message in place of a macro. (See Appendix F for details.)

String descriptors are 6-byte blocks of memory. The first 4 bytes contain the memory address of the string; the last 2 bytes contain the string size.

pbBuffer

is the memory address of a buffer to which the expanded message is written.

sBuffer

is the size of the buffer.

pCbRet

is the memory address of a word to which the actual length of the expanded message is returned.

Request Block

GetMsg is an object module procedure.

GetMsgUnexpanded (iMsg, pbBuffer, cbBuffer, pCbRet): ercType

Description

GetMsgUnexpanded retrieves a message from the message file and places the unexpanded message in memory, that is, it brings in an exact copy of the message text without expanding any macros which may be present. This operation generally is used in conjunction with the related operation, ExpandLocalMsg.

Procedural Interface

GetMsgUnexpanded (iMsg, pbBuffer, cbBuffer, pCbRet): ercType

where

iMsg

is the message number.

pbBuffer
cbBuffer

describe the buffer to which the message is to be copied.

pCbRet

is the memory address of a word to which the actual length of the unexpanded message is returned.

Request Block

GetMsgUnexpanded is an object module procedure.

This page intentionally left blank

GetMsgUnexpandedLength (iMsg, pcbRet): ercType

Description

GetMsgUnexpandedLength returns the length of a message in its unexpanded form. This operation generally is used to determine the memory needed to allocate for a message.

Procedural Interface

GetMsgUnexpandedLength (iMsg, pcbRet): ercType

where

iMsg

is the message number (word).

pcbRet

is the memory address of a word to which the length of the unexpanded message is returned.

Request Block

GetMsgUnexpandedLength is an object module procedure.

This page intentionally left blank

GetNlsDateName (pNlsTableArea, iDateNameTable, iDateName, pbRet, cbMax, pcbRet): ercType

Description

GetNlsDateName returns a date name string that displays the names of the months and the weekdays, as well as the strings, "Am," "Pm," "Noon," and "Midnight."

Date name strings are referenced by the following index values of *iDateName*:

1.	sunday	13.	june
2.	monday	14.	july
3.	tuesday	15.	august
4.	wednesday	16.	september
5.	thursday	17.	october
6.	friday	18.	november
7.	saturday	19.	december
8.	january	20.	am
9.	february	21.	pm
10.	march	22.	noon
11.	april	23.	midnight
12.	may	24.	output template #

The maximum length of a date name string is 20 bytes.

Two tables of the above date name strings are provided in the default set of NLS tables. Table number 1 contains full names as shown above; table number 2 contains the corresponding abbreviations, for example, mon, jan, and n.

A date name table number (*iDateNameTable*) is passed into GetNlsDateName to select the desired table number. If the value passed is invalid (not 1 or 2), the table number defaults to 1.

As described earlier, *iDateName*, in the range of 1 through 23 indexes one of the date name strings. A value of 24 returns the index of the default template from the Date and Time formats table for formatting the date and time strings. A value of 0 for *iDateTime*, however, has the special effect described below.

NOTE: A 0 value for iDateName is used internally only by the NlsParseTime and ParseTime operations.

Caution: *The 0 value operation is for* **internal use only.**

When *iDateName* is 0, GetNlsDateName returns the first entry (0) in the NLS Date Name Translations table. The entry is 2 bytes long and indicates how the date should be interpreted.

The first byte describes how to interpret the arrangement of the month (MM), day (DD), and year (YY) in the date entered. The byte values and their descriptions are as follows:

Value	Description
0	MM/DD/YY
1	DD/MM/YY
2	YY/MM/DD

The second byte describes the mode in which the hour is entered. The byte values and their descriptions are as follows:

Value	Description
0	AM/PM or Military (24-hour) mode (for example, 4:00 PM or 16:00)
1	Military mode only

Procedural Interface

GetNlsDateName (pNlsTableArea, iDateNameTable, iDateName, pbRet, cbMax, pcbRet): ercType

where

pNlsTableArea

GetNlsDateName returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the American date name strings as listed above.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Date Name table.
- If *pNlsTableArea* is address of an alternate set of NLS tables, an answer is returned based on the NLS Date Name table in the alternate set of NLS tables.

iDateNameTable

is the index (1 or 2) of the desired date name table.

iDateName

is the index (a value in the range of 1 through 24) of the desired date name. A value 0 for *iDateName* is used internally only.

pbRet
cbMax

describe the memory area to which the date name string is returned. If *iDateName* is not a value in the range 1 through 23, status code 2704 ("Bad date/time index") is returned.

pcbRet

is the memory address of the word to which the number of bytes returned is placed.

Request Block

GetNlsDateName is an object module procedure.

GetNlsDateTimeTemplate (pNlsTableArea, iTemplate, pbRet, cbMax, pcbRet): ercType

Description

GetNlsDateTimeTemplate returns a specified date and time format template from the NLS Date and Time Formats table. (See Appendix E for details on the NLS date and time format templates.)

The application provides the table index number of the format template to be returned. In addition, the application specifies the memory address of the NLS Date and Time Formats table from which the template will be read.

GetNlsDateTimeTemplate can be used to pass a format template to the NlsFormatDateTime operation. NlsFormatDateTime can then use the template to create a string that contains the date and time. (See NlsFormatDateTime.)

For details on NLS, see "Native Language Support" in the *CTOS Operating System Concepts Manual*.

Procedural Interface

GetNlsDateTimeTemplate (pNlsTableArea, iTemplate, pbRet, cbMax, pcbRet): ercType

where

pNlsTableArea

is the memory address of the NLS Data and Time Formats table from which the template is returned. GetNlsDateTimeTemplate returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, American templates are returned.

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Date and Time Formats table.
- If *pNlsTableArea* is the address of an alternate set of NLS tables, an answer is returned based on the NLS Date and Time Formats table in the alternate set of NLS tables.

iTemplate

is the index number of the desired date and time template. If this value is FFFFh, the default template (as specified in Nls.sys) is used. If Nls.sys is not present, template number 5 is selected.

pbRet
cbMax

describe the memory area to which the template string is returned. If this memory area is too small, GetNlsDateTimeTemplate truncates the returned string.

pcbRet

is the memory address of a word where the number of bytes returned is placed.

Request Block

GetNlsDateTimeTemplate is an object module procedure.

GetNlsKeycapText (pNlsTableArea, iKeycap, psbKeyCapTextRet): ercType

Description

GetNlsKeycapText returns the key cap text to be displayed when reference to a labeled key is desired.

Key caps are referenced by the following index values:

1.	ACTION	12.	MARK
2.	BACK SPACE	13.	MOVE
3.	BOUND	14.	NEXT
4.	CANCEL	15.	NEXT PAGE
5.	CODE	16.	OVER TYPE
6.	COPY	17.	PREV PAGE
7.	DELETE	18.	RETURN
8.	FINISH	19.	SCROLL DOWN
9.	GO	20.	SCROLL UP
10.	HELP	21.	SHIFT
11.	LOCK	22.	TAB

Procedural Interface

GetNlsKeycapText (pNlsTableArea, iKeycap, psbKeyCapTextRet): ercType

where

pNlsTableArea

GetNlsKeycapText returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the American key caps as listed above.

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Key Cap Legends table.
- If *pNlsTableArea* is address of an alternate set of NLS tables, an answer is returned based on the NLS Key Cap Legends table in the alternate set of NLS tables.

iKeycap

is the index of the desired key cap.

psbKeyCapTextRet

is the memory address of the area to which the key cap text is returned. The first byte is the length of the returned string; subsequent bytes contain the characters in the string.

Request Block

GetNlsKeycapText is an object module procedure.

GetNlsTable (wTableID, pMem, sMem, pbFileName, cbFileName, pbPassword, cbPassword): ercType

Description

GetNlsTable searches for the specified Native Language Support (NLS) table in the file NLS.sys loaded during operating system initialization. (For details on NLS, see "Native Language Support," in the *CTOS Operating System Concepts Manual.)* If the table is found, the table is copied to the memory area provided by the caller. If, however, the table is not found, GetNlsTable opens the NLS file described by *pbFileName/cbFileName*, loads it into short-lived memory, and copies the specified NLS table from this file into the memory area provided. At the completion of this operation, the NLS file described by *pbFileName/cbFileName* is closed (if it was opened) and all memory allocated for the file is deallocated.

Procedural Interface

GetNlsTable (wTableID, pMem, sMem, pbFileName, cbFileName, pbPassword, cbPassword): ercType

where

wTableID

is a word value that identifies the table ID code of the NLS table to be copied to the memory area provided by the caller.

pMem
sMem

describe the area in the caller's memory into which the specified NLS table is to be copied.

pbFileName
cbFileName

describe the file specification of the NLS file to be opened and copied to the caller's memory if the file Nls.sys does not contain the specified table. The specification is a character string of the form {*Node*}[*VolName*]<*DirName*>*FileName*.

pbPassword
cbPassword

describe the volume, directory, or file password that authorizes access to the file specified by *pbFileName/cbFileName*.

Request Block

GetNlsTable is an object module procedure.

GetOvlyStats (pStatRet, sStat): ercType

Description

GetOvlyStats returns the size of the largest overlay, the size of the second largest overlay, and the total size of all overlays in a program that uses the Virtual Code facility.

GetOvlyStats returns this information in a block of the following form:

cbOvMax	word, size of largest overlay
cbOv2Max	word, size of second largest overlay
cParMax	word, total of all overlays

To get a good estimate of minimum and miximum size of the segment to obtain, call GetOvlyStats prior to allocating the overlay zone.

Although you can choose a static size for your overlay zone, it is better practice to have your program determine dynamically what the overlay zone size should be. GetOvlyStats provides one such formula for setting the maximum and minimum size of the overlay zone.

Procedural Interface

GetOvlyStats (pStatRet, sStat): ercType

where

pStatRet

is the memory address of the block of information returned.

sStat

is the size of the returned block.

Request Block

GetOvlyStats is an object module procedure.

GetPartitionExchange (userNumPartition, pExchRet): ercType

NOTE: GetPartitionExchange is documented for backwards compatibility with older programs only. New applications should use the Intercontext Message Service (ICMS) rather than this operation for communicating with applications in other partitions. (For details, see "Interprocess Communication" in the CTOS Operating System Concepts Manual.)

Description

GetPartitionExchange returns the exchange number established by the SetPartitionExchange operation. The exchange number is used to communicate with an application program executing in another application partition.

Procedural Interface

GetPartitionExchange (userNumPartition, pExchRet): ercType

where

userNumPartition

is the user number returned from a CreatePartition or a GetPartitionHandle operation. A 0 specifies the application partition in which the client is executing.

pExchRet

is the memory address of a 16-bit word where the exchange is returned.

Request Block

sExchRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	184
12	userNumPartition	2	
14	pExchRet	4	
18	sExchRet	2	2

GetPartitionHandle (pbPartitionName, cbPartitionName,
pUserNumPartitionRet): ercType

Description

GetPartitionHandle returns the user number for the specific application partition name.

Procedural Interface

GetPartitionHandle (pbPartitionName, cbPartitionName,
pUserNumPartitionRet): ercType

where

pbPartitionName
cbPartitionName

describe the partition name (up to 12 characters).

pUserNumPartitionRet

is the memory address of the word where the user number is returned.

Request Block

sUserNumPartitionRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	177
12	pbPartitionName	4	
16	cbPartitionName	2	
18	pUserNumPartitionRet	4	
22	sUserNumPartitionRet	2	2

GetPartitionStatus (userNumPartition, statusCode, pStatusRet, sStatusMax):
ercType

Description

GetPartitionStatus returns status information about the specified application partition and the application program currently executing in it.

Procedural Interface

GetPartitionStatus (userNumPartition, statusCode, pStatusRet, sStatusMax):
ercType

where

userNumPartition

is the user number returned from a CreatePartition or a GetPartitionHandle operation. A 0 specifies the application partition within which the client is executing.

statusCode

specifies the status code. The status items and values are as follows:

Code	Item	Size (Bytes)
0	Partition Descriptor	33
1	Extended Partition Descriptor	172
2	Batch Control Block	1548
3	Application System Control Block	280
4	Partition Descriptor Address	4
5	Partition Information Block	12

For the format of each of these structures, see Chapter 4, "System Structures."

pStatusRet
sStatusMax

describe the memory area to which the status information is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	185
12	userNumPartition	2	
14	statusCode	2	
16	pStatusRet	4	
20	sStatusMax	2	

This page intentionally left blank

GetPartitionSwapMode (userNumPartition, pSwapModeRet): ercType

Description

GetPartitionSwapMode returns the swap mode established by the SetPartitionSwapMode operation.

Procedural Interface

GetPartitionSwapMode (userNumPartition, pSwapModeRet): ercType

where

userNumPartition

is the user number (word) returned from a CreatePartition or GetPartitionHandle operation. A 0 specifies the application partition in which the client is executing.

pSwapModeRet

is the memory address of a word where the swap mode is returned. See the SetPartitionSwapMode operation for definitions of possible swap modes.

Request Block

sSwapModeRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	364
12	userNumPartition	2	
14	pSwapModeRet	4	
18	sSwapModeRet	2	2

GetPAscb (ppASCBRet): ercType

Description

GetPAscb returns the address of the Application System Control Block (ASCB) of the application partition in which the application program is executing. (The format of this structure is shown in "Application System Control Block" in Chapter 4, "System Structures.")

Procedural Interface

GetPAscb (ppASCBRet): ercType

where

ppASCBRet

is the 4-byte memory address of the returned ASCB address.

Request Block

GetPAscb is a system-common procedure.

This page intentionally left blank

GetPNlsTable (pNlsTableArea, wTableID, ppNlsTableRet): ercType

Description

GetPNlsTable returns the memory address of a Native Language Support (NLS) table located within any NLS table area specified by *pNlsTableArea*. Use GetPNlsTable only if you are writing an NLS operation like NlsCollate, for example, which calls this operation to obtain the address of the NLS Collating Sequence Table.

All of the NLS operations call GetPNlsTable.

If *pNlsTableArea* is 0, GetPNlsTable calls GetPStructure with a structCode of 25 to obtain the base address, if present, of the set of NLS tables loaded during operating system initialization. (See GetPStructure for details.) From this address, GetPNlsTable obtains the address of the desired table and returns it to the calling program.

If, however, the set of NLS tables is not present, GetPNlsTable returns status code 7 ("Procedures not available to implement"). In this case, the operations default to algorithmic methods or use linked-in defaults.

If the requested table does not exist, status code 13500 ("No such table") is returned.

For a description of all the NLS tables, see "Native Language Support," in the *CTOS Operating System Concepts Manual*.

Procedural Interface

GetPNlsTable (pNlsTableArea, wTableID, ppNlsTableRet): ercType

where

pNlsTableArea

is either 0 (meaning use the NLS tables loaded at boot time), or is the address of an alternate set of NLS tables.

wTableID

is the word identifying the table ID code. 0-32767 are reserved; 32768-65535 are available for use.

ppNlsTableRet

is the 4-byte memory address to which the memory address of the NLS table is to be returned.

Request Block

GetPNlsTable is an object module procedure.

GetProcInfo (pMySlotRet, pTypeRet, pOrdinalNumRet, pbProcessorIdRet, cbProcessorIdMax, pcbProcessorIdRet): ercType;

Caution: *This operation works on shared resource processor hardware only. Do not use it on workstation hardware.*

Description

GetProcInfo returns the processor identifier (ID) of the SRP processor on which the caller is running. The format of the processor ID is *ttnn* where *tt* is the board type (such as FP, CP, TP, DP, SP, and so forth) and *nn* is the two-digit processor ordinal number (in decimal).

For example, if this operation is called in the second Cluster Processor, the results are as follows:

Result	Value
MySlotRet	<some value, such as 075h>
TypeRet	00Ch
NumberRet	01
ProcessorIdRet	CP01
cbProcessorIdRet	4

If this operation is called in the third File Processor, the results are as follows:

Result	Value
MySlotRet	<some value, such as 51h>
TypeRet	00Ah
OrdinalNumRet	02
ProcessorIdRet	FP02
cbProcessorIdRet	4

Procedural Interface

GetProcInfo (pMySlotRet, pTypeRet, pOrdinalNumRet, pbProcessorIdRet, cbProcessorIdMax, pcbProcessorIdRet): ercType;

where

pMySlotRet

is the memory address of a byte where the slot number of the board is returned. (See the *CTOS System Administration Guide* for details on slot numbers.)

pTypeRet

is memory address of a byte to which the processor type is returned. Processor types conform to the hardware type codes defined in the CPU Description Table (CDT). (For the format of the CDT, see "CPU Description Table" in Chapter 4, "System Structures,")

pOrdinalNumRet

is memory address of a byte to which the number identifying the processor's position within the shared resource processor (*processor ordinal number*) is returned. To determine the processor ordinal number, view the main enclosure from the rear. Count processors of the same type from left to right (ordinal numbers increase). The first processor of the desired type has ordinal number zero. If the system has a second enclosure (an expansion enclosure to the left of the main enclosure when viewed from the rear), continue counting processors from left to right within the expansion enclosure. Continue the count through any further enclosures in the same manner. (For an illustration of processor ordinal numbering, see "Understanding Hardware" in the *CTOS Sysem Administration Guide*.)

pbProcessorIdRet
cbProcessorIdMax

describe the memory area to which the processor ID is returned.

pcbProcessorIdRet

is the memory address of a byte to which the length of the processor ID (in bytes) is returned.

Request Block

GetProcInfo is a system-common procedure.

This page intentionally left blank

GetPSCounters (sh, pbBlockID, cbBlockID, pbCountRet, cbCountRet):
ercType

Description

GetPSCounters is another name for the PSGetCounters operation. See the description of PSGetCounters for details.

This page intentionally left blank

GetPStructure (structCode, userNum, ppStructRet): ercType

Description

GetPStructure returns the memory address of an operating system structure. This operation replaces the System-Common Address Table (SCAT) and facilitates system structure access by programs executing in protected mode.

This operation returns the address of the desired system structure. If the requested structure number is invalid or disabled, the operation returns an error.

When this operation is used to obtain a user-number specific structure for another user number (such as an ASCB), the returned memory address is valid only while that user number remains in memory. The address becomes invalid if the user number is swapped to disk or terminates.

For example, assume this operation is used to retrieve the address of the ASCB for a program running in another partition, and that program is later swapped to disk. The address returned by this operation before the swap occurred now points to something entirely different. In protected mode, using the address causes a protection fault. In real mode, using the address has unpredictable results.

Procedural Interface

GetPStructure (structCode, userNum, ppStructRet): ercType

where

structCode

identifies the operating system structure. The valid structure codes and their descriptions follow. By convention, the first 40 structure code numbers are identified using decimal notation. All structure code numbers greater than 40 are identified using hexadecimal notation.

Code	Structure†	Description
0	expardesc*	Extended Partition Descriptor.
1	Operating System Character Map*	
2	vcb	Video Control Block.
3	ascb	Application System Control Block.
4	vlpb	Variable Length Parameter Block.
5	bcb	Batch Control Block.
6	Type Ahead Buffer*	
7	rgpVidMemLine	Array of addresses specifying where each video line is stored in memory.
8	rgLineMap	Array of user numbers specifying which video lines are owned by which users.
9	rgContextStatus*	Partition swap status.
10	rgUserReadCount*	
11	rgpDebuggerState*	
12	graphicsInfo*	
13	ecb*	Event Control Block.

*This structure is for **internal use only** and is subject to change in future operating system releases.

†A pointer to the structure is returned.

Code	Structure†	Description
14	I/O Ports Structure	See "Port Structure" in Chapter 4, System Structures.
15	userNumVid	Current user number writing to the real screen.
16	System-Common Procedure Dispatch Table	
17	SbIBusIDTable	A 1-byte count of installed I-Bus IDs, followed by the IDs themselves.
18	Kernel Primitive Dispatch Table	
19	rgprgrqExchg*	Array of 17 addresses of arrays of service exchanges indexed by request number Is supported by operating systems prior to CTOS/XE 3.0. Code number 40 replsces this on CTOS/XE 3.0 and higher systems. The 17-element arrays referred by this code and by codes 22 through 24 correspond to the 17 request levels (0-0FFF, 1000-1FFF, ... 0E000–0EFFF, 0F000-0FFDF, and 0FFE0-0FFFF).
20	rgprcLookup*	
21	rgprcLookupBase*	
22	rgrcMax*	Array of 17 words containing the maximum number of request codes used in each of the 16 request levels.

*This structure is for **internal use only** and is subject to change in future operating system releases.

†A pointer to the structure is returned.

Code	Structure†	Description
23	rgprglclServiceCode*	Array of 17 addresses of arrays of local service codes indexed by request number.
24	rgprgNetRouting*	Array of 17 addresses of net routing information, indexed by request code. Is supported by operating systems prior to CTOS/XE 3.0. Code number 40 replsces this on CTOS/XE 3.0 and higher systems.
25	nlsTables	Base of the system-common native language support (NLS) table area.
26	screenStuff*	Base of tables used by installable video drivers.
27	cdt*	CPU Description Table (shared resource processor).
29	SbEncryptionKey	The key used by the system for password-encoding.
32	rgVidMemLineReal	Array of addresses specifying where each video line is stored in memory. These addresses are valid only for application programs executing in real mode on an operating system executing in protected mode (Rmos). Application programs should not use this code.
33	getpIntSwtbl	Interrupt information.
34	DiskStat*	Disk statistics table, used by the file system.
35	Stat*	Kernel process and exchange statistics table.

*This structure is for **internal use only** and is subject to change in future operating system releases.

†A pointer to the structure is returned.

Code	Structure†	Description
36	SbScratchVolumeSpec	The character string name of the system scratch volume.
37	OEM Common Data Area	A 32-byte data segment (with its own selector) reserved for use by OEMs. Note that exceeding the segment limit will cause a segment violation.
38	Vf*	Global operating system flags. (See "Operating System Flags" in Chapter 4, "System Structures.")
39	VideoConfig*	A 32-byte table, defined by the video
40	prgprgRouting*	Request Routing table. (See "Inter-CPU Communication in the *CTOS Operating System Concepts Manual*.) Is supported on CTOS/XE 3.0 and higher systems by real mode and protected mode processors but not by RMOS programs. Replaces the Exchange table, the Net Routing table, and the shared resource processor routing table.
44	sbGlobalUserName	The 31-character string containing the sign-on user name. The first byte contains the length of the string.
45	ParKbdInfo	The Partition Keyboard Information Structure. (This structure is described in Chapter 4, "System Structures.")

*This structure is for **internal use only** and is subject to change in future operating system releases.

†A pointer to the structure is returned.

The *vcb* and *ascb* codes that follow are for the caller's user number. In these cases, the value of the *userNum* parmeter to this operation is ignored. The remaining codes are for system-wide structures. All numbers are in hexadecimal.

Structures that are offsets such as *oRgOVhb* and *oRgOVhb* may not be in the same segment. Be sure to use the segment base address of the original pointer of a structure to build a pointer to that structure.

Code	Structure†	Description
240h	sysTime	System date/time structure.
244h	vcb	Video Control Block (All workstations).
248h	rgSysError	System Error Status Block.
24Ch	pPcbRun	Memory address of the Process Control Block of the currently running process.
250h	ascb	Application System Control Block.
254h	sbVerRun	Version is a character string whose length is defined by its own first byte.
264h	cLinkBlks*	Word containing the count of available link blocks.
268h	oLinkBlkAvail*	Offset to first link block on a linked list of link blocks available.
270h	rgKbdEncode	Keyboard Mapping Table. (The U.S. default contents are shown in Appendix B.)
274h	oRgExch*	Offset to exchange zero.
278h	oRgLinkBlk*	Offset to the first link block.

*This structure is for **internal use only** and is subject to change in future operating system releases.

†A pointer to the structure is returned.

Code	Structure†	Description
27Ch	oRgODcb	Offset to the first entry in the array of offsets to the Device Control Blocks (DCBs).
280h	oRgOFcb*	Offset to the first entry in the array of offsets in the File Control Blocks (FCBs).
284h	oRgOUcb*	Offset to the first entry in the array of offsets to the User Control Blocks (UCBs).
288h	oRgOVhb*	Offset to the first entry in the array of offsets to the Volume Home Blocks (VHBs).
28Ch	oRgPcb	Offset to the first Process Control Block (PCB).
290h	oRgPTrb*	Offset to the first entry in the array of memory addresses of active timer request blocks.
29Ch	runQ*	Queue of ready-to-run Process Control Blocks.
2A0h	SaSemiHighest	Reserved. See code number 2C8h.
2A4h	SaSemiLowest	Reserved. See code number 2C8h.
2A8h	SaTempHighest	Reserved. See code number 2C8h.
2ACh	SaTempLowest	Reserved. See code number 2C8h.
2B0h	SaEndMemory	Reserved. See code number 2C8h.
2B4h	bootBlock	A 16-word array that contains the information passed to the operating system by the bootstrap ROM.

*This structure is for **internal use only** and is subject to change in future operating system releases.

†A pointer to the structure is returned.

Code	Structure†	Description
2B8h	sbBootDevName	Character string containing the name of the device from which the operating system was bootstrapped. If a workstation is bootstrapped across a cluster, the string contains a workstation identifier of the type WSxxx, where xxx is a workstation number. The first byte of the string contains the byte count of the string.
2BCh	sbCrashDumpDevName	Character string containing the name of the device to which the operating system crash dump was written. The first byte of the string contains the byte count of the string.
2C0h	hdcCntlBlk*	Hard Disk (Winchester) Control Block.
2C4h	floppyCntlBlk*	Floppy Disk Control Block.
2C8h	configBlk	System Configuration Block.
2D4h	sysDeviceNum	System device number.
2E4h	extCntlReg	A byte that is a copy of the contents of the external control register. (The control register is write-only).
2E8h	grPortVal	The last byte written to the graphics Multibus port.
2ECh	fontRamBuf	Obsolete.
2F0h	HdwState	Same as code number 2E8h.
2F8h	rgpVidMemLine	Same as code number 7.

*This structure is for **internal use only** and is subject to change in future operating system releases.

†A pointer to the structure is returned.

userNum

is the user number returned from a CreatePartition or a GetPartitionHandle operation. If *userNum* is 0, then the user number of the calling program is used.

ppStructRet

is the memory address to which the address of the operating system structure should be returned.

Request Block

GetPStructure is a system-common procedure.

This page intentionally left blank

GetQMStatus (wQueueType, fHealthCheck, pStatusRet, sStatusMax):
ercType

Description

The GetQMStatus operation provides a method for interrogating the Queue Manager about usage statistics, as well as the queues of the specified type.

Procedural Interface

GetQMStatus (wQueueType, fHealthCheck, pStatusRet, sStatusMax):
ercType

where;

wQueueType

is the type of queue (an integer less than or equal to 255).

fHealthCheck

if TRUE, check all queues of the requested type for structural integrity.

pStatusRet
sStatusMax

describe the memory area where the Queue Manager status information is returned.

The format of the memory area described above is as follows:

Offset	Field	Size (Bytes)	Contents
0	cQueueFetch	4	total number of queue fetches requested since the last statistics check.
4	cacheCatch	2	percentage of queue fetches that were fielded by cache.
6	nQueues	2	number of queues.
8	rgQueue	54	array that contains nQueues queue entries.

Each queue entry contains fields as described below:

Offset	Field	Size (Bytes)	Contents
0	qH	2	queue handle created by the Queue Manager for this queue.
2	sbQueueName	51	queue name string with size prefix.
53	fContaminated	1	set only if the *fHealthCheck* flag was set and the queue needs re-setting.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32868
12	wQueueType	2	
14	fHealthCheck	1	
15	reserved	1	
16	pStatusRet	4	
20	sStatusMax	2	

This page intentionally left blank

GetRouteTable (wCase, pbTableRet, cbTableMax): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. It is supported on shared resource processors only.*

Description

GetRouteTable returns processor routing information in the memory area provided. The structure and contents of the routing table information returned depend on the value specified for *wCase*. If the memory area provided is not large enough to contain all the routing information to be returned, the information is truncated.

Procedural Interface

GetRouteTable (wCase, pbTableRet, cbTableMax): ercType

where

wCase

is a word value that indicates the type and contents of the routing table to be returned. The values of *wCase* and the meaning of the formatted information returned by each are described below:

Value	Description
0	GetRouteTable fills in a caller-supplied array of words with the cluster line numbers and the corresponding processor slot numbers for up to the maximum number of cluster lines on a shared resource processor (currently 32). The value of array element 0 is always 0. Starting with array element 1, the line number is returned in the high-order byte, and the corresponding slot number in the low-order byte. All line and slot numbers are returned in ascending numberic order in the array, for example,

Element	Line Number	Slot Number
1	0	62h
2	1	62h
3	0	73h
4	1	73h

Value	Description
1	GetRouteTable fills in a caller-supplied array of bytes with the processor slot numbers corresponding to file processor indexes. The value of array element 0 is the slot number of the master File Processor. Array element 1 contains the slot number of the file processor with index 1; array element 2, the slot number of the file processor with index 2; and so forth up to the maximum number of indexes.

pbTableRet
cbTableMax

describe the memory area to which the routing table information is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	rtCode	1	0
2	nReqPbCb	0	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	312
12	wCase	2	
14	reserved	4	
18	pbTableRet	4	
22	cbTableMax	2	

This page intentionally left blank

GetRsLfa (pRswa, pLfaRet): ercType

Description

GetRsLfa returns the logical file address at which the next input/output operation will occur for the open RSAM file identified by the memory address of the Record Sequential Work Area.

Procedural Interface

GetRsLfa (pRswa, pLfaRet): ercType

where

pRswa

is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

pLfaRet

is the memory address to which the current logical file address is to be returned.

Request Block

GetRsLfa is an objec module procedure.

This page intentionally left blank

GetScsiInfo(wModule, wBus, wFmt, pBufInfo, sBufInfo, pCbRet): ercType

GetScsiInfo returns information about the devices connected to the specified SCSI bus.

This operation has been renamed ScsiQueryInfo. For a complete description, see the description of ScsiQueryInfo in this chapter. Refer to Appendix J, "Operation Data," to determine which operating systems support the new operation name.

This page intentionally left blank

GetSegmentLength (pSegment, pcbRet): ercType

Caution: GetSegmentLength is only useful to programs executing in protected mode. If called from a real mode program, this procedure returns 65,535 (maximum segment length in bytes).

Description

GetSegmentLength accepts a memory address and returns the length of the segment. If the pointer is NULL or is an invalid pointer, GetSegmentLength returns status code 405 ("Bad selector name").

Procedural Interface

GetSegmentLength (pSegment, pcbRet): ercType

where

pSegment

is a memory address of a segment for which the segment length is to be returned.

pcbRet

is a memory address to a word where the length of the segment is to be returned.

Request Block

GetSegmentLength is an object module procedure.

This page intentionally left blank

GetServerMsg (iMsg, pSdRet): ercType

Description

GetServerMsg is used to extract a particular message (string) from a message file previously initialized by OpenServerMsgFile.

(See the related operation, OpenServerMsgFile.)

Procedural Interface

GetServerMsg (iMsg, pSdRet): ercType

where

iMsg

is the message number of the desired message.

pSdRet

is the address of a 6-byte block of memory that describes the desired message. The memory address of the message string is returned in the first 4 bytes, and its size is returned in the last 2 bytes.

Request Block

GetServerMsg is an object module procedure.

This page intentionally left blank

GetSlotInfo (bCode, pbSlotArrayRet, cbSlotArrayRet, psProcessorCountRet): ercType

Caution: *This operation works on shared resource processor hardware only. Do not use it on workstation hardware.*

Description

GetSlotInfo determines the slot numbers of a group of related processors in a shared resource processor. (For details on slot numbering, see the *CTOS System Administration Guide*.)

GetSlotInfo fills a caller-supplied array with the slot numbers of all of the processors of a specified type, and it returns the number of processors of that type. GetSlotInfo may be used to find the slot number of the master File Processor and is the preferred method of determining what processors are attached to the system bus.

If the caller supplies an array too small to hold all of the slot numbers, as many entries as possible are placed in the array and status code 211 ("BadBufferSize) is returned; the number of processors is correctly returned.

Procedural Interface

GetSlotInfo (bCode, pbSlotArrayRet, cbSlotArrayRet, psProcessorCountRet): ercType

where

bCode

takes on one of the following values:

bCode	Processor Type
1	File Processor
2	Terminal Processor
3	Cluster Processor
4	Applications Processor (not supported on CTOS/XE, Version 3.0 and higher systems)
5	Storage Processor (not coupled with an SC)
6	Data Processor (SP with SC)
7	Master File Processor
8	Reserved
9	File Processor Class (FP, DP, GP+SI)
10	Tape Processor Class (DP, SP, GP+SI)
11	Reserved
12	General Processor (with no CI or SI)
13	GP+SI (GP with SCSI interface)
14	GP+CI (GP with communications interface)
15	General Processor Class (GP, GP+SI, GP+CI)
16	Cluster Processor Class (CP, GP, GP+SI, GP+CI)

If *bCode* is a value other than one of the above, status code 154 ("No such code") is returned. Other values of *bCode* are reserved for future expansion.

pbSlotArrayRet
cbSlotArrayRet

describe the memory area where the slot numbers of the processors specified by *bCode* are returned.

psProcessorCountRet

is the memory address of a word to which the number of processors of a given type is returned.

Request Block

GetSlotInfo is a system-common procedure.

This page intentionally left blank

GetStamFileHeader (pbFileSpec, cbFileSpec, pbPassword, cbPassword, pFileHeaderRet): ercType

Description

GetStamFileHeader copies the file header of an RSAM, DAM, or ISAM file into the specified area. (See Chapter 4, "System Structures," for the format of the standard file header. See the *CTOS Indexed Sequential Access Method (ISAM II) Programming Reference Manual* for more information on ISAM.)

Procedural Interface

GetStamFileHeader (pbFileSpec, cbFileSpec, pbPassword, cbPassword, pFileHeaderRet): ercType

where

pbFileSpec
cbFileSpec

describe a character string specifying the name of the file whose header is to be read.

pbPassword
cbPassword

describe a character string specifying the password authorizing the requested file access.

pFileHeaderRet

describes the memory area to which the file header is returned. The memory area must be at least 512 bytes long and word aligned.

Request Block

GetStamFileHeader is an object module procedure.

This page intentionally left blank

GetStandardErcMsg (erc, pBuf, cbBuf, pcbRet, iMsgDefault): ercType

Description

GetStandardErcMsg accepts a status code (erc) and returns the corresponding error message into a caller supplied buffer.

GetStandardErcMsg first searches for the standard error message in the binary file, [Sys]<Sys>Erc.bin. If Erc.bin does not contain the message, GetStandardErcMsg will attempt to use the message specified by *iMsgDefault* from the caller's current message file (if one is opened). The caller's message must contain the macro %0N, which would be replaced by the status code. For example,

:500: "Error %0N (See documentation.)"

If *iMsgDefault* is 500 and the status code is 203, GetStandardErcMsg would return the message

Error 203 (See documentation.)

If, for any reason, GetStandardErcMsg fails to retrieve a message from the caller's message file, or if *iMsgDefault* is 0, GetStandardErcMsg would return the message

"Error <status code>"

(For details on creating messages, see "Native Language Support," in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

GetStandardErcMsg (erc, pBuf, cbBuf, pcbRet, iMsgDefault): ercType

where

erc

is the status code.

pBuf

is the memory address of the caller supplied buffer.

cbBuf

is the size of the buffer in bytes.

pcbRet

is the memory address of a word into which the size of the message is placed.

iMsgDefault

specifies a message in the caller's current message file.

Request Block

GetStandardErcMsg is an object module procedure.

GetSysCmdInfo (pbCommand, cbCommand, pbFileNameRet, pcbFileNameRet, pbCmdNameRet, pcbCmdNameRet, pbCaseRet, pwNumParamsRet, ppCmdRecRet): ercType

Description

GetSysCmdInfo returns the following information from a specified command name or command name abbreviation:

full command name
run file
case value
number of parameters
memory address of a command record

GetSysCmdInfo searches the command file opened with InitSysCmds. If the command does not exist, status code 2611 ("No such command") is returned. If more than one command can be referenced by the specified command abbreviation, status code 2612 ("Not unique command") is returned.

Procedural Interface

GetSysCmdInfo (pbCommand, cbCommand, pbFileNameRet, pcbFileNameRet, pbCmdNameRet, pcbCmdNameRet, pbCaseRet, pwNumParamsRet, ppCmdRecRet): ercType

where

pbCommand
cbCommand

describe the command name or an abbreviation of the command name to be referenced.

pbFileNameRet
cbFileNameRet

describe the memory area where the run file name for the command is returned.

pbCmdNameRet
cbCmdNameRet

describe the memory area where the full command name is returned.

pcbCmdNameRet

is the memory address of a word to which the length of the command name is returned.

pbCaseRet

is the 2-byte memory address where the command case is returned.

pwNumParamsRet

is the memory address of a word where the number of parameters in the command form is returned.

ppCmdRecRet

is the memory address to which the address of the command record is returned.

Request Block

GetSysCmdInfo is an object module procedure.

GetUcb (pUcbRet, sUcbMax): ercType

Description

GetUcb copies the User Control Block (UCB) for the current user number to the specified area. If the specified area is not large enough to hold the requested information, the information is truncated. (See Chapter 4, "System Structures," for the structure of the UCB.)

Procedural Interface

GetUcb (pUcbRet, sUcbMax): ercType

where

pUcbRet
sUcbMax

describe the memory area to which the User Control Block is copied.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	27
12	reserved	6	
18	pUcbRet	4	
22	sUcbMax	2	

This page intentionally left blank

GetUserFileEntry (pNlsTableArea, pBuffer, sBuffer, pbFieldName, cbFieldName, pbValue, sValueMax, pcbValueRet): ercType

Description

GetUserFileEntry opens the operator's .user file, searches for a ":FieldName:" string entry, closes the file, and returns the string value it finds just following the trailing colon of the entry. Unlike the LookUpString operation, which starts searching at the current file location, GetUserFileEntry starts its search at the beginning of the file and assumes that each field name in the file is unique.

If a Native Language Support (NLS) table area is present at system boot, the name of the .user file can be nationalized. See the description of *pNlsTableArea*. (For details on NLS, see "Native Language Support," in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

GetUserFileEntry (pNlsTableArea, pBuffer, sBuffer, pbFieldName, cbFieldName, pbValue, sValueMax, pcbValueRet): ercType

where

pNlsTableArea

The user file suffix .user is constructed differently based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, the suffix for the user file is .user.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, the suffix is constructed based on the value in the NLS Strings table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, the suffix is constructed based on the value in the NLS Strings table in the alternate set of NLS tables.

pBuffer
sBuffer

describe a buffer to be used in the search. *sBuffer* (word) must be a multiple of 512.

pbFieldName
cbFieldName

describe the field name entry to be searched for. The colons preceding and following the entry should not be given as part of the field name.

pbValue
sValueMax

describe the memory area where the string following the field name trailing colon is copied.

pcbValueRet

is the memory address of a 2 byte area where the count of bytes in the string is returned.

Request Block

GetUserFileEntry is an object module procedure.

This page intentionally left blank

GetUserLocation (userNum, pbInfoRet, cbInfoMax): ercType

NOTE: Programs that may run on earlier versions of the operating system should use this object module procedure rather than the request QueryUserLocation, which returns the same information. Doing so will avoid application termination with status code 31 ("No such request code").

Description

GetUserLocation returns information about the specified user number in the memory area provided. If the memory area is too small to contain all the information returned by this operation, the information is truncated.

The information returned by this operation can be used, for example, to determine if the user number is executing on a specified processor of a shared resource processor. (See the description of the related operation, GetBoardInfo.)

Procedural Interface

GetUserLocation (userNum, pbInfoRet, cbInfoMax): ercType

where

userNum

is the user number (word) returned from a CreatePartition or a GetPartitionHandle operation. If userNum is 0, the user number is that of the calling program.

pbInfoRet
cbInfoMax

describe the memory area at which the information described below is returned.

Offset	Field	Size (bytes)	Description
0	bCode	1	(See below.)
1	localUserNum	2	Is the local user number associated with the *iCpu* (described below).
3	iCpu	2	Is the number of the CPU upon which the specified user number is executing. (The user number consists of a *localUserNum* and an *iCpu*. For workstations, the value of *iCpu* is 0.)

bCode is one of the following values:

Value	Meaning
0	The user number is on this CPU.
1	The user number is on this shared resource processor (shared resource processors only).
2	The user number is on a cluster workstation.
3	The user number is located over the CT network.
4	The user number is not valid.

Request Block

GetUserLocation is an object module procedure.

GetUserNumber (pUserNumberRet): ercType

Description

GetUserNumber allows a process to determine its own user number (also known as a partition handle).

For compatibility with previous versions of the operating system, GetUserNumber returns 1 if executed in an application partition when there are no other application partitions in memory. Otherwise GetUserNumber returns the actual user number.

If a program needs to know its true user number when executing in the only application partition in memory, it can call GetPStructure to access its Process Control Block (PCB). It can then retrieve its user number from its PCB.

Procedural Interface

GetUserNumber (pUserNumberRet): ercType

where

pUserNumberRet

is the memory address of a word where the user number of the inquiring process is returned.

Request Block

GetUserNumber is a system-common procedure.

This page intentionally left blank

GetUserStatus (userNumStatus, pStatusRet, sStatusMax): ercType

Description

GetUserStatus copies user number status information to the specified memory area. If this area is not large enough to hold the status information, the information is truncated.

Procedural Interface

GetUserStatus (userNumStatus, pStatusRet, sStatusMax): ercType

where

userNumStatus

is the user number for which the status is returned.

pStatusRet
sStatusMax

describe the memory area to which status information is returned. The status information has the following format:

Offset	Field	Size (bytes)	Description
0	statusCode	1	A value of 0 means user is at the same workstation where the GetUserStatus request was issued. A value of 1 means the user is at a cluster workstation, and the GetUserStatus request was issued at a server workstation. A value of 2 means the user is at a remote node.
1	sbUserName	31	User name. Returned only if the user is not at a remote node.
33	sbNodeName	13	Node name. Returned only if the user is at a remote node.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	rtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	272
12	userNumStatus	2	
14	reserved	4	
18	pStatusRet	4	
22	sStatusMax	2	

This page intentionally left blank

GetVhb (pbDevSpec, cbDevSpec, pVhbRet, sVhbMax): ercType

Description

GetVhb copies the Volume Home Block (VHB) of the specified device to the specified memory area. If the specified area is not large enough to hold the requested information, the information is truncated.

GetVhb does not require a password. To avoid security violations, 0's are returned in the *volPassword* field of the Volume Home Block.

(See Chapter 4, "System Structures" for the format of the VHB.)

Procedural Interface

GetVhb (pbDevSpec, cbDevSpec, pVhbRet, sVhbMax): ercType

where

pbDevSpec
cbDevSpec

describe a character string of the form {Node}[DevName] or {Node}[VolName]. Square brackets are optional for the device name. The distinction between uppercase and lowercase is not significant in matching device names.

pVhbRet
sVhbMax

describe the memory area.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	15
12	reserved	6	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pVhbRet	4	
28	sVhbMax	2	

GetWsUserName (WSNum, pWSUserNameRet, sWSUserNameRetMax): ercType

Caution: *This operation is for compatibility with prior operating system releases only and may not be available in future releases.*

Description

GetWsUserName returns the SignOn user name at the specified cluster workstation. QueryWsNum can be called to obtain the *WSNum* parameter value for this operation.

GetClusterStatus should be used instead of this operation to obtain the user names of active workstations. (See GetClusterStatus.)

Procedural Interface

GetWsUserName (WSNum, pWSUserNameRet, sWSUserNameRetMax): ercType

where

WSNum

is the workstation identification number returned by QueryWsNum. (See QueryWsNum for the format of *WSNum*.)

pWSUserNameRet

is the memory address of the first byte of the buffer where the user name is returned. The first byte of the character string is the size (maximum of 31).

sWSUserNameRetMax

is the size of the buffer.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	202
12	WSNum	2	
14	reserved	4	
18	pWSUserNameRet	4	
22	sWSUserNameRetMax	2	

USER'S COMMENT SHEET

We welcome your comments and suggestions. They help us improve our manuals. Please give specific page and paragraph references whenever possible.

Does this manual provide the information you need? Is it at the right level? What other types of manuals are needed?

Is this manual written clearly? What is unclear?

Is the format of this manual convenient in arrangement, in size?

Is this manual accurate? What is inaccurate?

Name _____ Date _____

Title _____ Phone _____

Company Name/Department _____

Address _____

City _____ State _____ Zip Code _____

Thank you. All comments become the property of Unisys Corporation

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 1807 SAN JOSE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

UNISYS
Multimedia Product Information 9–007
2700 N 1st St
San Jose CA 95134–2028

Fold Here

Tape Please Do Not Staple Tape

43574714-110